JÜLICH
FORSCHUNGSZENTRUM

CUBE 4.3.2 – User Guide

Generic Display for Application Performance Data

June 14, 2015

The Scalasca Development Team
scalasca@fz-juelich.de

Copyright

Copyright © 1998–2015 Forschungszentrum Jülich GmbH, Germany

Copyright © 2009–2015 German Research School for Simulation Sciences GmbH, Jülich/Aachen, Germany

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the names of Forschungszentrum Jülich GmbH or German Research School for Simulation Sciences GmbH, Jülich/Aachen, nor the names of their contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Contents

Copyright iii

1 Cube User Guide 1
- 1.1 Abstract . . . 1
- 1.2 Introduction . . . 1
- 1.3 Command line options . . . 3
- 1.4 Environment variables . . . 3
- 1.5 Using the Display . . . 4
- 1.6 Plugins . . . 26
- 1.7 Performance Algebra and Tools . . . 46

2 CUBE4 API 63
- 2.1 Creating CUBE Files . . . 63

3 Appendix 81
- 3.1 File format of statistics files . . . 81

Bibliography 83

1 Cube User Guide

1.1 Abstract

CUBE is a presentation component suitable for displaying performance data for parallel programs including MPI and OpenOpenMP applications. Program performance is represented in a multi-dimensional space including various program and system resources. The tool allows the interactive exploration of this space in a scalable fashion and browsing the different kinds of performance behavior with ease. CUBE also includes a library to read and write performance data as well as operators to compare, integrate, and summarize data from different experiments. This user manual provides instructions of how to use the CUBE display, how to use the operators, and how to write CUBE files.

The version 4 of CUBE implementation has an incompatible API and file format to preceding versions.

1.2 Introduction

CUBE (CUBE Uniform Behavioral Encoding) is a presentation component suitable for displaying a wide variety of performance data for parallel programs including MPI[1] and OpenOpenMP[2] applications. CUBE allows interactive exploration of the performance data in a scalable fashion.cube_ Scalability is achieved in two ways: hierarchical decomposition of individual dimensions and aggregation across different dimensions. All metrics are uniformly accommodated in the same display and thus provide the ability to easily compare the effects of different kinds of program behavior.

CUBE has been designed around a high-level data model of program behavior called the *cube performance space*. The CUBE performance space consists of three dimensions: a metric dimension, a program dimension, and a system dimension. The *metric dimension* contains a set of metrics, such as communication time or cache misses. The *program dimension* contains the program's call tree, which includes all the call paths onto which metric values can be mapped. The *system dimension* contains the items executing in parallel, which can be processes or threads depending on the parallel programming model. Each point (m,c,s) of the space can be mapped onto a number representing the actual measurement for metric m while the control flow of process/thread s was executing call path c . This mapping is called the *severity* of the performance space.

Each dimension of the performance space is organized in a *hierarchy*. First, the metric dimension is organized in an inclusion hierarchy where a metric at a lower level is a sub-

set of its parent. For example, communication time is a subset of execution time. Second, the program dimension is organized in a call-tree hierarchy. However, sometimes it can be advantageous to abstract away from the hierarchy of the call tree, for example if one is interested in the severities of certain methods, independently of the position of their invocations. For this purpose CUBE supports also flat call profiles, that are represented as a flat sequence of all methods. Finally, the system dimension is organized in a multi-level hierarchy consisting of the levels: machine, SMPnode, process, and thread.

CUBE also provides a *library* to read and write instances of the previously described data model in the form of a .CUBEXfile (which is a TAR TARfile `anchor.xml` inside of the CUBEXenvelope. The data part contains the actual severity numbers to be mapped onto the different elements of the performance space and stored in binary format in various files inside of the CUBEXenvelope.

The *display* component can load such a file and display the different dimensions of the performance space using three coupled tree browsers (figure1.1). The browsers are connected in such a way that you can view one dimension with respect to another dimension. The connection is based on *selections*: in each tree you can select one or more nodes. For example, in Figure1.1 the `Execution` metric, the `adi` call path node, and `Process 0` are selected. For each tree, the selections in the trees on its left-hand-side (if any) restrict the considered data: The metric nodes aggregate data over all call path nodes and all system items, the call tree aggregates data for the `Execution metric` over all system nodes, and each node of the system tree shows the severity for the *Execution* metric of the `adi` call path node for this system node.

If the CUBE file contains topological information, the distribution of the performance metric across the topology can be examined using the *topology view*. Furthermore, the display is augmented with a source-code display that shows the position of a call site in the source code.

As performance tuning of parallel applications usually involves multiple experiments to compare the effects of certain optimization strategies, CUBE includes a feature designed to simplify cross-experiment analysis. The *CUBE algebra*[4] is an extension of the framework for multi-execution performance tuning by Karavanic and Miller[3] and offers a set of operators that can be used to compare, integrate, and summarize multiple CUBE data sets. The algebra allows the combination of multiple CUBE data sets into a single one that can be displayed and examined like the original ones.

In addition to the information provided by plain CUBE files a statistics file can be provided, enabling the display of additional statistical information of severity values. Furthermore, a statistics file can also contain information about the most severe instances of certain performance patterns -- globally as well as with respect to specific call paths. If a trace file of the program being analyzed is available, the user can connect to a trace browser (i.e., Vampir or Paraver) and then use CUBE to zoom their timelines to the most severe instances of the performance patterns for a more detailed examination of the cause of these performance patterns.

The following sections explain how to use the CUBE display, how to create CUBE files, and how to use the algebra and other tools.

1.3 Command line options

To invoke GUI for CUBE profile exploration one uses command:

```
cube [-disable-plugins] [-preload] [-lastN] [-help] filename.cubex
```

A list of main options:

`-preload` All data is read at the begining and held in memory

`-help` Display list of command line options

%todo

1.4 Environment variables

CUBE provides the option of displaying an online description for entries in the metric tree via a context menu. By default, it will search for the given HTML description file on all the mirror URLs specified in the CUBE file. In case there is no Internet connection, the Qt-based CUBE GUI can be configured to also search in a list of local directories for documentation files. These additional search paths can be specified via the environment variable `CUBE_DOCPATH` as a colon-separated list of local directories, e.g.,

```
CUBE_DOCPATH=/opt/software/doc:/usr/local/share/doc
```

Note that this feature is only available in the Qt-based GUI and **not** in the older wxWidgets-based one.

To prevent CUBE from trying to load the HTML documentation via HTTP or HTTPS mirror URLs (e.g., in restricted environments were outbound connections are blocked by a firewall and the timeout is taking very long), the environment variable `CUBE_DISABLE_HTTP_DOCS` can be set to either `1`, `yes` or `true`.

CUBE C++ library allows to control the way it loads the data using the environment variable `CUBE_DATA_LOADING`. Following values are possible:

1. **keepall** - data is loaded on demand and kept in memory to the end of lyfecycle of the Cube object.
2. **preload** - all data is loaded during the metric initialization and kept in memory to the end of lyfecycle of the Cube object.

3. **manual** - Application should request and drop the data sets explicitly. No correctness check is performed. Therefore one has to use this strategy with care.
4. **lastn** - Only `N` last used data rows are kept in memory. `N` is specified via environment variable `CUBE_NUMBER_ROWS`

1.5 Using the Display

This section explains how to use the CUBE-QT display component. After installation, the executable "`cube`" can be found in the specified directory of executables (specifiable by the "prefix" argument of configure, see the CUBE Installation Manual). The program supports as an optional command-line argument the name of a cube file that will be opened upon program start.

After a brief description of the basic principles, different components of the GUIwill be described in detail.

1.5.1 Basic Principles

The CUBE-QT display has three tree browsers, each of them representing a dimension of the performance space (figure1.1). Per default, the left tree displays the metric dimension, the middle tree displays the program dimension, and the right tree displays the system dimension. The nodes in the metric tree represent metrics. The nodes in the program dimension can have different semantics depending on the particular view that has been selected. In Figure1.1 , they represent call paths forming a call tree. The nodes in the system dimension represent machines, nodes, processes, or threads from top to bottom.

Each node is associated with a value, which is called the *severity* and is displayed simultaneously using a numerical value as well as a colored square. Colors enable the easy identification of nodes of interest even in a large tree, whereas the numerical values enable the precise comparison of individual values. The sign of a value is visually distinguished by the *relief* of the colored square. A raised relief indicates a positive sign, a sunken relief indicates a negative sign.

Users can perform two basic types of actions: selecting a node or expanding/collapsing a node. In the metric tree in figure1.1 , the metric `Execution` is selected. *Selecting* a node in a tree causes the other trees on its right to display values for that selection. For the example of figure1.1 , the metric tree displays the total metric values over all call tree and system nodes, the call tree displays values for the `Execution` metric over all system entities, and the system tree for the `Execution` metric and the `adi` call tree node. Briefly, a tree is always an aggregation over all selected nodes of its neighboring trees to the left.

Collapsed nodes with a subtree that is not shown are marked by a [+] sign, *expanded*

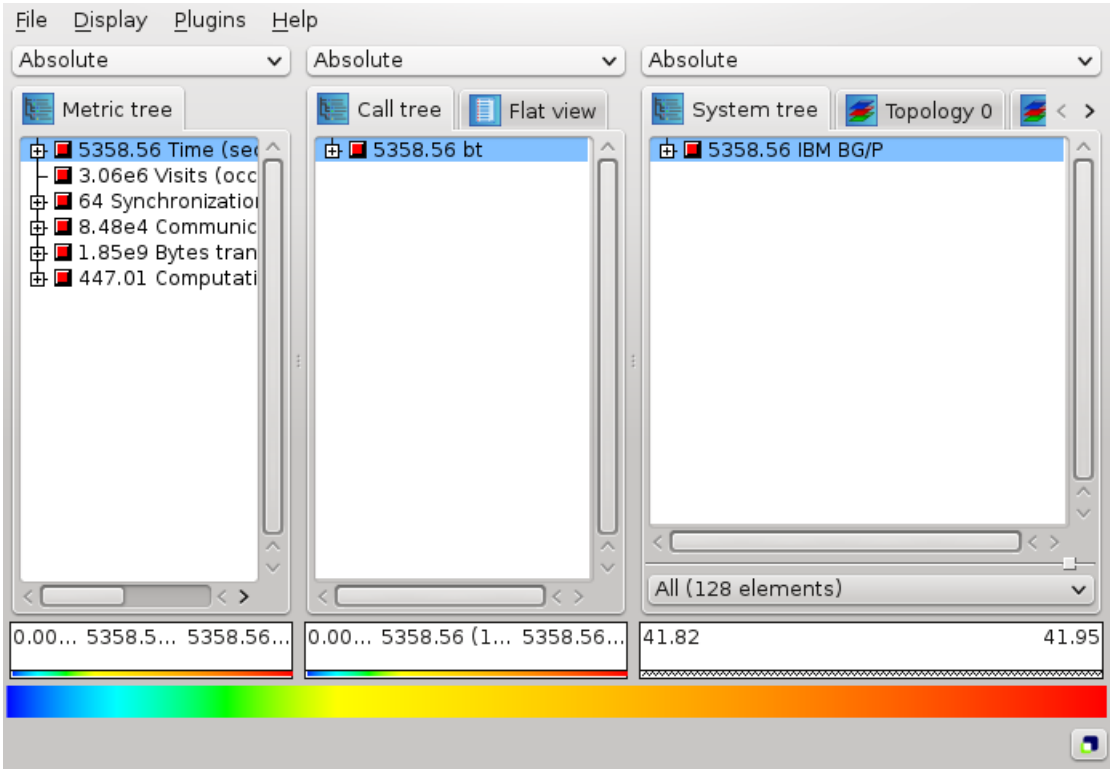

Figure 1.1: CUBE display window

nodes with a visible subtree by a [-] sign. You can expand/collapse a node by left-clicking on the corresponding [+]/[-] signs. Collapsed nodes have *inclusive* values, i.e., their severity is the sum of the severities over the whole collapsed subtree. For the example of Figure1.1 , the `Execution` metric value 3496.10 is the total time for all executions. On the other hand, the displayed values of expanded nodes are their *exclusive* values. E.g., the expanded `Execution` metric node in Figure1.2 shows that the program needed 2839.54 seconds for execution other than `MPI`.

Note that expanding/collapsing a selected node causes the change of the current values in the trees on its right-hand side. As explained above, in our example in Figure1.1 the call tree displays values for the `Execution` metric over all system entities. Since the `Execution` node is collapsed, the call tree severities are computed for the whole `Execution` metric's subtree. When expanding the selected `Execution` node, as shown in Figure1.2 , the call tree displays values for the `Execution` metric without the `MPI` metric.

1.5.2 GUI Components

The GUIconsists (from top to bottom) of

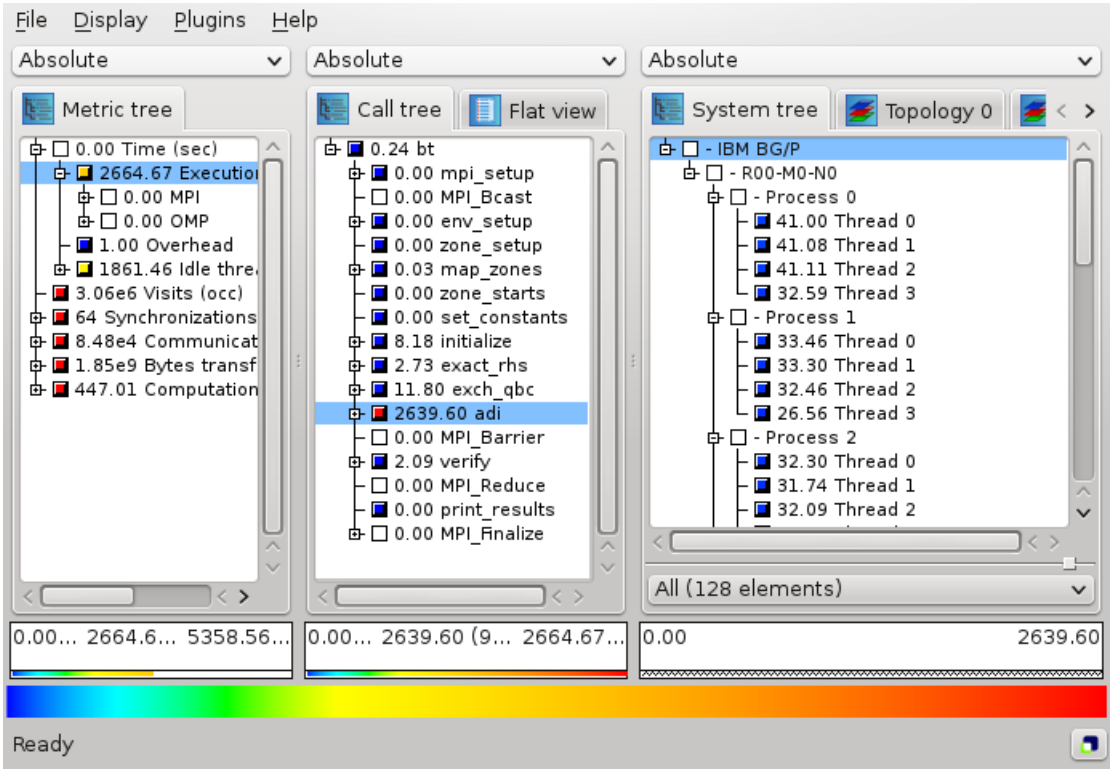

Figure 1.2: CUBE display window with expanded metric node ”Execution”

- a menu bar,
- three value mode combo boxes,
- three resizable panes each containing some tabs,
- three selected value information widgets,
- a color legend, and
- a status bar.

The three resizable panes offer different views: the metric, the call, and the system pane. You can switch between the different tabs of a pane by left-clicking on the desired tab at the top of the pane. Note that the order of the panes can be changed (see the description of the menu item *Display* ⇒ *Dimension order* in Section1.6.4.2).

The metric pane provides only the metric tree browser. The call pane offers a call tree browser and a flat call profile. The system pane has a system tree browser. Tree browsers also provide a context menu.

1.5.2.1 menu

The menu bar consists of four menus: a file menu, a display menu, a plugin menu and a help menu. Some menu functions also have a keyboard shortcut, which is written besides the menu item's name in the menu. E.g., you can open a file with Ctrl+O without going into the menu. A short description of the menu items is visible in the status bar if you stay for a short while with the mouse above a menu item.

1. **File:** The file menu offers the following functions:
   a) **Open (Ctrl+O):** Offers a selection dialog to open a CUBE file. In case of an already opened file, it will be closed before a new file gets opened. If a file got opened successfully, it gets added to the top of the recent files list (see below). If it was already in the list, it is moved to the top.
   b) **Save as (Ctrl+S):** Offers a selection dialog to save a copy of a CUBE file. Opened CUBE file stays loaded in cube.
   c) **Close (Ctrl+W):** Closes the currently opened CUBE file. Disabled if no file is opened.
   d) **Open external:** Opens a file for the external percentage value mode (see Section1.5.2.2).
   e) **Close external:** Closes the current external file and removes all corresponding data. Disabled if no external file is opened.

      %deprecated
   f) **Connect to trace browser:** This menu item is only visible if a CUBE file with a corresponding statistics file, containing information about the most severe instances of certain performance patterns, is open and CUBE was configured for remote trace browsing. In this case, it offers to connect to a trace browser (i.e., Vampir or Paraver) to examine the behaviour of the program around the most severe pattern instances. For an in-depth explanation of this feature see subsection1.6.6.2.
   g) **Settings:** This menu item offers the saving, loading, and the deletion of settings. There are two types of settings, the global settings and the experiment settings.

      The global settings don't depend on the loaded cube file and are saved in a system specific format. These settings e.g. store the appearance of the application like the widget sizes, color and precision settings, the order of panes, etc. The default settings are automatically saved on exit and restored at startup, but it is also possible to save several settings under different names.

      The experiment settings depend on the loaded cube file. They allow to store e.g. which tree nodes are selected and which are expanded, the selected value mode etc. These settings are saved next to the opened cube file in the file

cubebasename.ini. When saving experiment settings, the global settings are also saved in the .ini file. Like global settings, the default experiment settings are automatically saved and restored, but another behaviour may be chosen in the Settings menu. If the experiment settings toolbar is enabled, named settings can be selected and be saved in the .ini file.

h) **Screenshot:** The function offers you to save a screen snapshot in a PNG file. Unfortunately the outer frame of the main window is not saved, only the application itself.

i) **Quit (Ctrl+Q):** Closes the application.

j) **Recent files:** The last 5 opened files are offered for re-opening, the top-most being the most recently opened one. A full path to the file is visible in the status bar if you move the mouse above one of the recent file items in the menu.

2. **Display:** The display menu offers the following functions:

a) **Dimension order:** As explained above, CUBE has three resizable panes. Initially the metric pane is on the left, the call pane is in the middle, and the system pane is on the right-hand side. However, sometimes you may be interested in other orders, and that is what this menu item is about. It offers all possible pane orderings. For example, assume you would like to see the metric and call values for a certain thread. In this case, you could place the system pane on the left, the metric pane in the middle, and the call pane on the right, as shown in Figure1.3 . Note that in panes to the left of the metric pane no meaningful valuescan be presented, since they miss a reference metric; in this case values are specified to be undefined, denoted by a "-" (minus) sign.

b) **General coloring:** Allows for selection of color maps and changing of color settings in a new dialog. In the configuration dialog, the `Ok` button applies the settings to the display and closes the dialog, the `Apply` button applies the settings to the display, and `Cancel` cancels all changes since the dialog was opened (even if "Apply" was pressed in between) and closes the dialog.

i. **default:** Default color map for Cube. The configuration dialog is show in Figure1.4 . At the top of the dialog you see a color legend with some vertical black lines, showing the position of the color scale start, the colors cyan, green, and yellow, and the color scale end. These lines can be dragged with the left mouse button, or their position can also be changed by typing in some values between 0.0 (left end) and 1.0 (right end) below the color legend in the corresponding spins.

The different coloring methods offer different functions to interpolate the colors at positions between the 5 data points specified above.

With the upper spin below the coloring methods you can define a threshold percentage value between 0.0 and 100.0, below which colors are

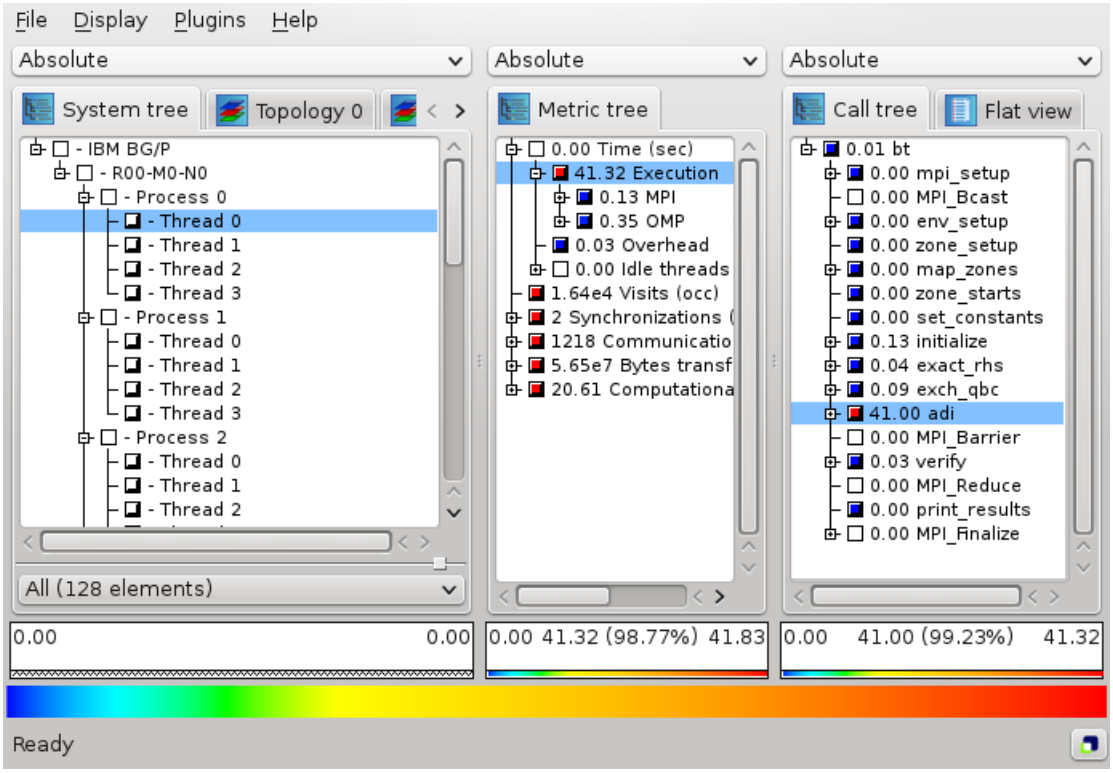

Figure 1.3: Modified pane order via the menu ”Display ⇒ Dimension order”

lightened. The nearer to the left end of the color scale, the stronger the lightening (with linear increase).

With the spin at the bottom of the dialog you can define a threshold percentage value between 0.0 and 100.0 , below which values should be colored white.

ii. **Advanced Color Maps** Cube plugin which provides additional color maps. The configuration dialogs are presented in Figure 1.5 . For every color map, the plot allows for change of data accepted by color map and one can do that using left and right marker, by dragging the marker or providing exact position through a double click near the marker value (new dialog will appear). The default color for values out of range is grey.

One can change colors of scheme (for some color maps) and color for values out of range. Double mouse click on proper part of the plot opens a dialog with selection of RGB color. Additionally, one can adjust the plot marker or reset to default values through the context menu.

Currently the plugin adds four different sets of color maps:

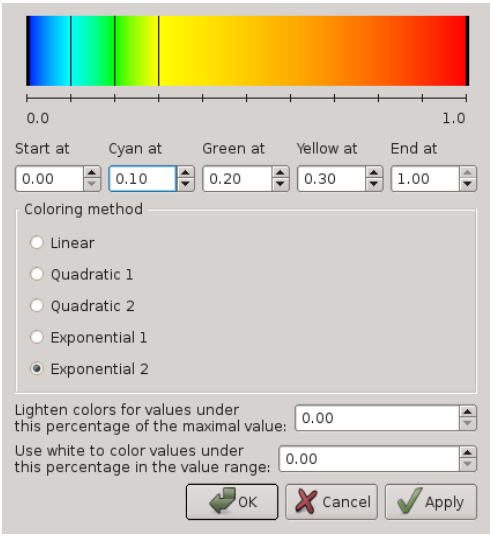

Figure 1.4: The color dialog opened via the menu ”Display ⇒ General coloring”

A. **Sequential:** Scheme is defined by starting and ending color with linear or exponential interpolation between them. Predefined schemes provide simple interpolation from one color to pure white. Middle marker allows for subtle change of interpolation.

B. **Divergent:** This scheme is defined by an interpolation from starting to ending color, but with a critical value between them, depicted with the pure white. The position of critical point can be set with the middle marker.

C. **Cubehelix:** Scheme designed primarily for display of astronomical intensity images. The coloring is based on distribution from black to white, with R, G and B helixes giving additional deviations. Cubehelix is defined by four parameters:

   Start colour - starting value for color, floating-point number between 0.0 and 3.0. R = 1, G = 2, B = 0

   Rotations - floating-point number of R -> G -> B rotations from the start to the end. Negative value corresponds to negative direction of rotation.

   Hue - non-negative value which controls saturation of the scheme, with pure greyscale for hue equal to 0.

   Gamma factor - non-negative value which configures intensity of colours. Values below one emphasizes low intensity values and creates brighter color scheme. Values above one emphasizes high intensity values and generates darker color map.

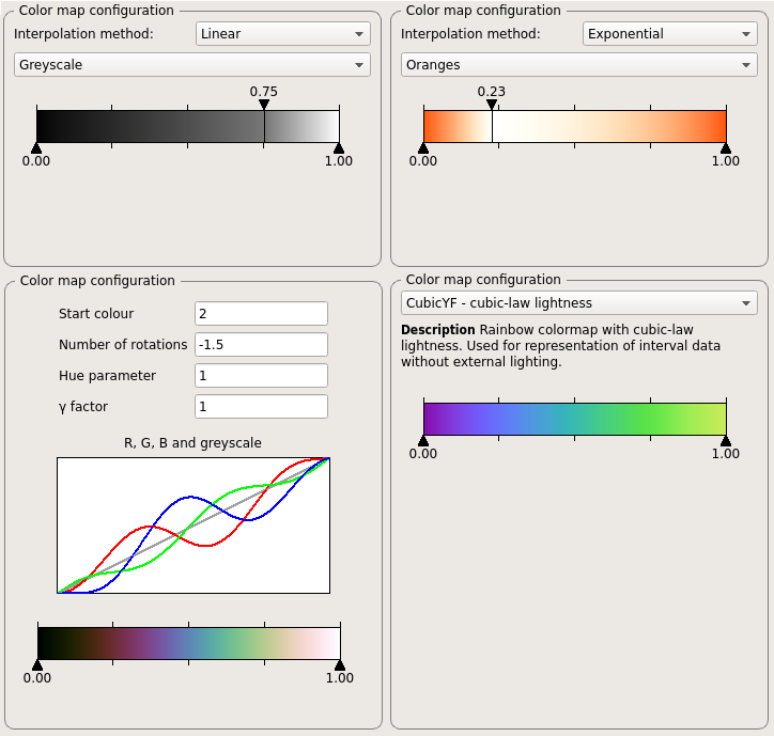

Figure 1.5: The examples of configuration for Advanced Color Maps. Upper row, starting from left: sequential, divergent; lower row, starting from left: cubehelix, improved rainbow.

Reference: Green, D. A., 2011, 'A colour scheme for the display of astronomical intensity images', Bulletin of the Astronomical Society of India, 39, 289.

D. **Improved rainbow colormap:** Set of color maps based on original jet (rainbow) scheme, but with different lightness distribution. The goal behind these schemes is to provide map with more balanced perception, which is poor for original jet, mainly because of sharp changes in lightness. These maps doesn't provide any possibility for configuration.

Reference: Perceptually improved colormaps, MATLAB Central

c) **Precision:** Activating this menu item opens a dialog for precision settings (see Figure1.6). Besides `Ok` and `Cancel`, the dialog offers an `Apply` button, that applies the current dialog settings to the display. Pressing `Cancel` undoes *all* changes due to the dialog, even if you already pressed `Apply` previously, and closes the dialog. `Ok` applies the settings and closes the dialog.

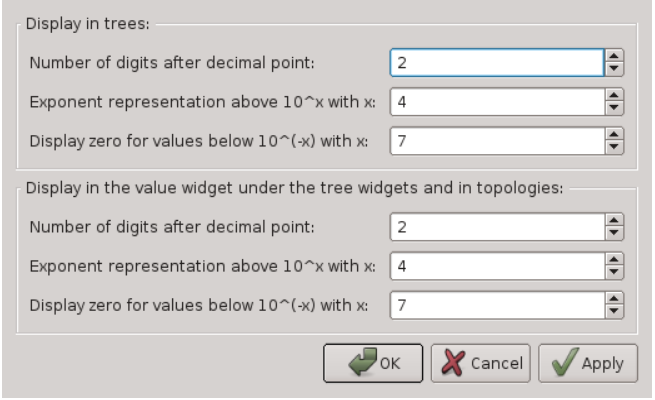

Figure 1.6: Display ⇒ Precision

It consists of two parts: precision settings for the tree displays, and precision settings for the selected value info widgets and the topology displays. For both formats, three values can be defined:

i. **Number of digits after the decimal point:** As the name suggests, you can specify the precision for the fraction part of the values. E.g., the number 1.234 is displayed as 1.2 if you set this precision to 1, as 1.234 if you set it to 3, and as 1.2340 if you set it to 4.

ii. **Exponent representation above 10^x with x:** Here you can define above which threshold scientific notation should be used. E.g., the value 1000 is displayed as 1000 if this value is larger then 3 and as $1e3$ otherwise.

iii. **Display zero values below 10^{-x} with x:** Due to inexact floating point representation, it often happens that users wish to round down values very near by zero to zero. Here you can define the threshold below which this rounding should take place. E.g., the value 0.0001 is displayed as 0.0001 if this value is larger than 3 and as zero otherwise.

d) **Trees:** This menu item offers two sub-items:

i. **Font:** Here you can specify the font, the font size (in pt), and the line spacing for the tree displays (see Figure1.7). The `Ok` button applies the settings to the display and closes the dialog, the `Apply` button applies the settings to the display, and `Cancel` cancels all changes since the dialog was opened (even if `Apply` was pressed in between) and closes the dialog.

ii. **Selection marking:** Here you can specify if selected items in trees should be marked by a blue background or by a frame.

e) **Optimize width:** Under this menu item CUBE offers widget rescaling such that the amount of information shown is maximized, i.e., CUBE optimally

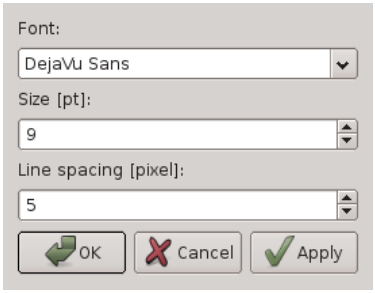

Figure 1.7: The font dialog opened via the menu ”Display ⇒ Trees ⇒ Font”

distributes the available space between its components. You can chose if you would like to stick to the current main window size, or if you allow to resize it.

3. **Plugins:** The plugin menu allows the user to define which plugins are laoded. For each loaded plugin, a submenu is added. The submenu contains a menu item to enable or disable the plugin and the plugin may add additional menu items.
   a) **Initial activation settings:** Opens a dialog to define which plugins should be loaded.
4. **Help:** The help menu provides help on usage and gives some information about CUBE.
   a) **Getting started:** Opens a dialog with some basic information on the usage of CUBE.
   b) **Mouse and keyboard control:** Lists mouse and keyboard controls as given in Section1.6.7.
   c) **What’s this?:** Here you can get more specific information on parts of the CUBE GUI. If you activate this menu item, you switch to the “What’s this?” mode. If you now click on a widget, an appropriate help text is shown. The mode is left when help is given or when you press Esc.

      Another way to ask the question is to move the focus to the relevant widget and press Shift+F1.
   d) **About:** Opens a dialog with release information.
   e) **Selected metric description:** Opens a new window showing the description of the currently selected metric, equivalent to *Online description* in the metric tree context menu. Disabled if online information is unavailable.
   f) **Selected region description:** Opens a new window showing the description of the currently selected region, equivalent to *Online description* in the call-tree context menu. Disabled if online information is unavailable.

1.5.2.2 Value modes

Each tree view has its own value mode combobox, a drop-down menu above the tree, where it is possible to change the way the severity values are displayed.

The default value mode is the **Absolute** value mode. In this mode, as explained below, the severity values from the CUBE file are displayed. However, sometimes these values may be hard to interpret, and in such cases other value modes can be applied. Basically, there are three categories of additional value modes.

- The first category presents all severities in the tree as percentage of a reference value. The reference value can be the absolute value of a selected or a root node from the same tree or in one of the trees on the left-hand side. For example, in the **Own root percent** value mode the severity values are presented as percentage of the own root's (inclusive) severity value. This way you can see how the severities are distributed within the tree. All the value modes (Own root percent -- System selection percent) fall into this category.

  All nodes of trees on the left-hand side of the metric tree have undefined values. (Basically, we could compute values for them, but it would sum up the severities over all metrics, that have different meanings and usually even different units, and thus those values would not have much expressiveness.) Since we cannot compute percentage values based on undefined reference values, such value modes are not supported. For example, if the call tree is on the left-hand side, and the metric tree is in the middle, then the metric tree does not offer the **Call root percent** mode.

- The second category is available for system trees only, and shows the distribution of the values within hierarchy levels. E.g., the **Peer percent** value mode displays the severities as percentage of the maximal value on the same hierarchy depth. The value modes (Peer percent -- Peer distribution) fall into this category.

- Finally, the **External percent** value mode relates the severity values to severities from another external CUBE file (see below for the explanation).

Depending on the type and position of the tree, the following value modes may be available:

1. **Absolute (default):** Available for all trees. The displayed values are the severity value as read from the cube file, in units of measurement (e.g., seconds). Note that these values can be negative, too, i.e., the expression "absolute" in not used in its mathematical sense here.
2. **Own root percent:** Available for all trees. The displayed node values are the percentage of their absolute values with respect to the absolute value of their root node in collapsed state.
3. **Metric root percent:** Available for trees on the right-hand side of the metric tree. The displayed node values are the percentage of their absolute values with respect to the absolute value of the collapsed metric root node. If there are several metric

roots, the root of the selected metric node is taken. Note, that multiple selection in the metric tree is possible within one root's subtree only, thus there is always a unique metric root for this mode.

4. **Metric selection percent:** Available for trees on the right-hand side of the metric tree. The displayed node values are the percentage of their absolute values with respect to the selected metric node's absolute value in its current collapsed/expanded state. In case of multiple selection, the sum of the selected metrics' values for the percentage computation is taken.
5. **Call root percent:** Available for trees on the right-hand side of the call tree. Similar to the metric root percent, but the call tree root instead of the metric tree root is considered. In case of multiple selection with different call roots, the sum of those root values is considered.
6. **Call selection percent:** Available for trees on the right-hand side of the call tree. Similar to the metric selection percent, percentage is computed with respect to the selected call node's value in its current collapsed/expanded state. In case of multiple selections, the sum of the selected call values is considered.
7. **System root percent:** Available for trees on the right-hand side of the system tree. Similar to the call root percent, the sum of the inclusive values of all roots of selected system nodes are considered for percentage computation.
8. **System selection percent:** Available for trees on the right-hand side of the system tree. Similar to the call selection percent, percentage is computed with respect to the selected system node(s) in its current collapsed/expanded state.
9. **Peer percent:** For the system tree only. The peer percentage mode shows the percentage of the nodes' inclusive absolute values relative to the largest inclusive absolute peer value, i.e., to the largest inclusive value between all entities on the current hierarchy depth. For example, if there are 3 threads with inclusive absolute values 100, 120, and 200, then they have the peer percent values 50, 60, and 100.
10. **Peer distribution:** For the system tree only. The peer distribution mode shows the percentage of the system nodes' inclusive absolute values on the scale between the minimum and the maximum of peer inclusive absolute values. For example, if there are 3 threads with absolute values 100, 120 and 200, then they have the peer distribution values 0, 20 and 100.
11. **External percent:** Available for all trees, if the metric tree is the left-most widget. To facilitate the comparison of different experiments, users can choose the external percentage mode to display percentages relative to another data set. The external percentage mode is basically like the metric root percentage mode except that the value equal to 100% is determined by another data set.

Note that in all modes, only the leaf nodes in the system hierarchy (i.e., processes or threads) have associated severity values. All other hierarchy levels (i.e., machines, nodes and eventually processes) are only used to structure the hierarchy. This means that their

severity is undefined---denoted by a "-" (minus) sign---when they are expanded.

1.5.2.3 System resource subsets

By default, all system resources (typically threads) are included when determining box-plot statistics. Other defined subsets can be chosen from the combobox below the box-plot, such as "Visited" threads which are only those threads that visited the currently selected callpath. The current subset is retained until another is explicitly chosen or a new subset is defined.

Additional subsets are defined from the system tree with the *Define subset* context menu using the currently selected threads via multiple selection (Ctrl+<left-mouse click>) or with the *Find Items* context menu selection option.

1.5.2.4 Tree browsers

A tree browser displays different hierarchical data structures in form of trees. Currently supported tree types are metric trees, call trees, flat call profiles, and system trees. The structure of the displayed data is common in all trees: The indentation of the tree nodes reflects the hierarchical structure. Expandable nodes, i.e., nodes with non-hidden children, are equipped with a [+]/[-] sign ([+] for collapsed and [-] for expanded nodes). Furthermore, all nodes have a color icon, a value, and a label.

The value of a node is computed, as explained earlier, basing on the current selections in the trees on the left-hand side and on the current value mode. The precision of the value display in trees can be modified, see the menu item *Display* ⇒ *Precision* in Section1.6.4.2. The color icon reflects the position of the node's value between 0.0 and a maximal value. These maximal value is the maximal value in the tree for the absolute value mode, or 100.0 otherwise. See the menu item *Display* ⇒ *General coloring* in Section1.6.4.2 and the context menu item *Min/max values* in the context menu description below for color settings.

A label in the metric tree shows the metric's name. A label in the call tree shows the last callee of a particular call path. If you want to know the complete call path, you must read all labels from the root down to the particular node you are interested in. After switching to the flat profile view (see below), labels in the flat call profile denote methods or program regions. A label in the system tree shows the name of the system resource it represents, such as a node name or a machine name. Processes and threads are usually identified by a rank number, but it is possible to give them specific names when creating a CUBE file. The thread level of single-threaded applications is hidden. Multiple root nodes are supported.

After opening a data set, the middle panel shows the call tree of the program. However, a user might wish to know which fraction of a metric can be attributed to a particular region (e.g., method) regardless of from where it was called. In this case, you can switch

from the call-tree view (default) to the flat-profile view (Figure1.8). In the flat-profile view, the call-tree hierarchy is replaced with a source-code hierarchy consisting of two levels: regions and their subroutines. Any subroutines are displayed as a single child node labeled *Subroutines*. A subroutine node represents all regions directly called from the region above. In this way, you are able to see which fraction of a metric is associated with a region exclusively, that is, without its regions called from there.

Tree displays are controlled by the left and right mouse buttons and some keyboard keys. The left mouse button is used to select or expand/collapse a node: You can expand/collapse a node by left-clicking on the attached [+]/[-] sign, and select it by left-clicking elsewhere in the node's line. To select multiple items, Ctrl+<left-mouse click> can be used. Selection without the Ctrl key deselects all previously selected nodes and selects the clicked node. In single-selection mode you can also use the up/down arrows to move the selection one node up/down. The right mouse button is used to pop up a context menu with node-specific information, such as online documentation (see the description of the context menu below).

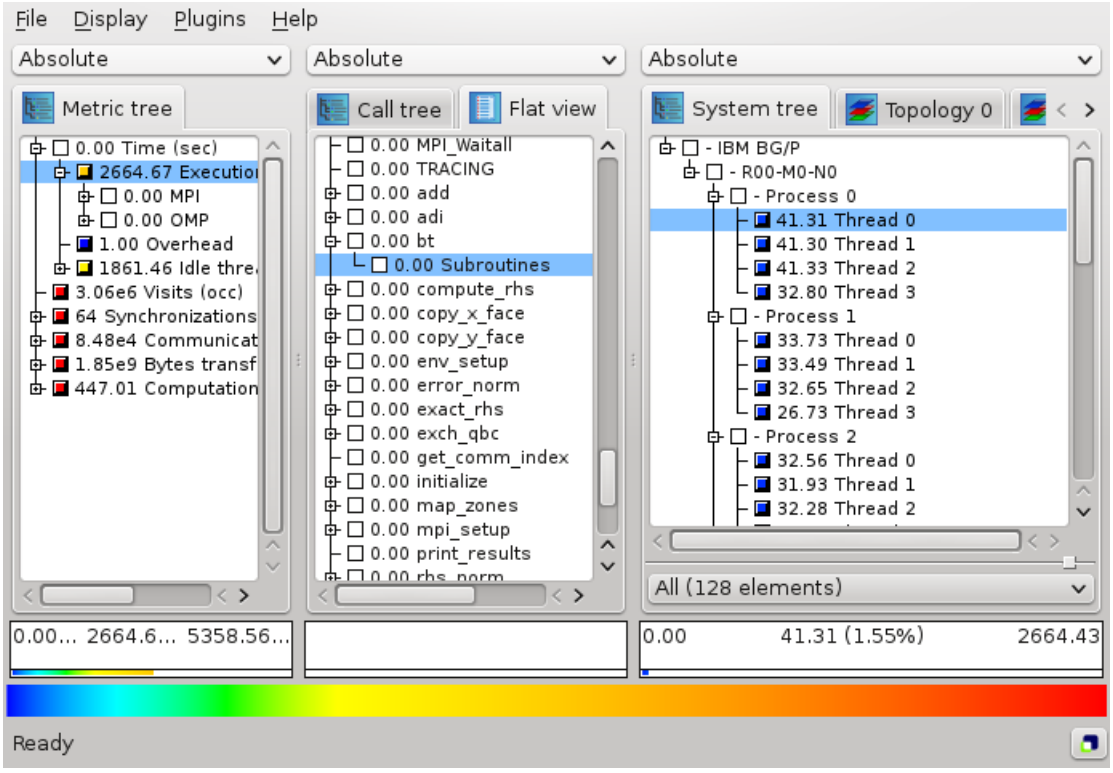

Figure 1.8: CUBE flat profile

Each tree has its own context menu which can be activated by a right mouse click within the tree's window. If you right-click on one of the tree's nodes, this node gets framed, and serves as a *reference node* for some of the menu items. If you click outside of tree

items, there is no refernce node, and some menu items are disabled.

The context menu consists, depending on the type of the tree, of some of the following items. If you move the mouse over a context menu item, the status bar displays some explanation of the functionality of that item.

1. **Collapse all:** For all trees. Collapses all nodes in the tree.
2. **Collapse subtree:** For all trees. Enabled only if there is a reference node. It collapses all nodes in the subtree of the reference node (including the reference node).
3. **Collapse peers:** For system trees only. Enabled only if there is a reference node. Collapses all peer nodes of the reference node, i.e., all nodes at the same hierarchy level.
4. **Expand all:** For all trees. Expands all nodes in the tree.
5. **Expand subtree:** For all trees. Enabled only if there is a reference node. Expands all nodes in the subtree of the reference node (including the reference node).
6. **Expand peers:** For system trees only. Enabled only if there is a reference node. Expands all peer nodes of the reference node, i.e., all nodes at the same hierarchy level.
7. **Expand largest:** For all trees. Enabled only if there is a reference node. Starting at the reference node, expands its child with the largest inclusive value (if any), and continues recursively with that child until it finds a leaf. It is recommended to collapse all nodes before using this function in order to be able to see the path along the largest values.
8. **Dynamic hiding:** Not available for metric trees. This menu item activates dynamic hiding. All currently hidden nodes get shown. You are asked to define a percentage threshold between 0.0 and 100.0. All nodes whose color position on the color scale (in percent) is below this threshold get hidden. As default value, the color percentage position of the reference node is suggested, if you right-clicked over a node. If not, the default value is the last threshold. The hiding is called dynamic, because upon value changes (caused for example by changing the node selection) hiding is re-computed for the new values. In other words, value changes may change the visibility of the nodes.

   a) **Redefine threshold:** This menu item is enabled if dynamic hiding is already activated. This function allows to re-define the dynamic hiding threshold as described above.

   During dynamic hiding, for expanded nodes with some hidden children and for nodes with all of its children hidden, their displayed (exclusive) value includes the hidden children's inclusive value. The percentage of the hidden children is shown in brackets next to this aggregate value.

9. **Static hiding:** Not available for metric trees. This menu item activates static hiding. All currently hidden nodes stay hidden. Additionally, you can hide and show nodes using the now enabled sub-items:

   a) **Static hiding of minor values:** Enabled only in the static hiding mode. As described under dynamic hiding, you are asked for a hiding threshold. All nodes whose current color position on the color scale is below this percentage threshold get hidden. However, in contrast to dynamic hiding, these hidings are static: Even if after some value changes the color position of a hidden node gets above the threshold, the node stays hidden.

   b) **Hide this:** Enabled only in the static hiding mode if there is a reference node. Hides the reference node.

   c) **Show children of this:** Enabled only in the static hiding mode if there is a reference node. Shows all hidden children of the reference node, if any.

   Like for dynamic hiding, for expanded nodes with some hidden children and for nodes with all of its children hidden, their displayed (exclusive) value includes the hidden children's inclusive value. The percentage of the hidden children is shown in brackets next to this aggregate value.

10. **No hiding:** Not available for metric trees. This menu item deactivates any hiding, and shows all hidden nodes.

11. **Find items:** For all trees. Opens a dialog to get a regular expression from the user. If the user called the context menu over an item, the default text is the name of the reference node, otherwise it is the last regular expression which was searched for.

    If *select items* is checked, items matching the regular expression also become selected.

    If *select items* is unchecked, all non-hidden nodes whose names contain the given text are marked with a yellow background, and all collapsed nodes whose subtree contains such a non-hidden node by a light yellow background. The current node found, that is initialized to the first found node, is marked by a distinguished yellow hue.

12. **Find next:** For all trees. Changes the current found node to the next found node. If you did not start a search yet, then you are asked for the regular expression to search for.

13. **Clear found items:** For all trees. Removes the background markings of the preceding find items.

14. **Define subset:** Only for system tree. Uses the currently selected system resources (e.g., from a preceding *Find items*) to create a new subset of all system resources (typically threads) with the provided name. This is added to the combobox at the bottom of the system tree and boxplot statistics panes, and becomes the currently active subset for which statistics are calculated.

15. **Info:** For all trees (for call trees under *Called region*). Gives some short information about the reference node. Disabled if there is no reference node or if no information is available for the reference node.

16. **Full Info:** For metric tree and call tree only. In the case of metric tree it lists a complete information about the selected metric. One gets information about display and unique name, data type, unit of measurements, kind of metric and CubePL expression if the metric is derived.

    In the case of call tree it lists a complete available information about the selected call path. One gets information about call path id (to use it with command line tools like `cube_dump`), region begining line, region ending line, region module, url with the online help and finally description of the region.

    Disabled if not clicked over metric item or call path item.

17. **Online description:** For metric trees and flat call profiles (for call trees see under *Called region*). Shows some (usually more extensive) online description for the reference node. For example, metrics might point to an online documentation explaining their semantics, or regions representing library functions might point to the corresponding library documentation. Disabled if there is no reference node or if no online information is available.

18. **Location:** For flat profiles only. Disabled if there is no reference node. Displays information about the module and position within the module (line numbers) where the method is defined.

19. **Source code:** For flat call profiles only (for call trees see *Call site* and *Called region* below). Disabled if there is no reference node. Opens an editor for displaying, editing, and saving the source code of the method/region to which the reference node refers. The begin and the end of the method/region are highlighted. If the specified source file is not found, you are asked to choose a file to open.

    The file is in a read-only mode per default. If you wish to edit the text, please uncheck the `Read only` box in the bottom left corner. For keyboard and mouse control, see Section1.6.7.

20. **Hide iterations:** Only visible for calltree items that are recognized or manually defined as loop (see "Set as loop" below). By activating, all children of the loop are hidden. The grandchildren are shown and its values for the different iterations are aggregated (see Figure1.9).

21. **Call site:** For call trees only. Enabled only if there is a reference node. Offers information about the caller of the reference node.

    a) **Location:** Displays information about the module and position within the module (line numbers) of the caller of the reference node.

    b) **Source code:** Opens an editor for displaying, editing, and saving the source code where the call for which the reference node stays for happens. The

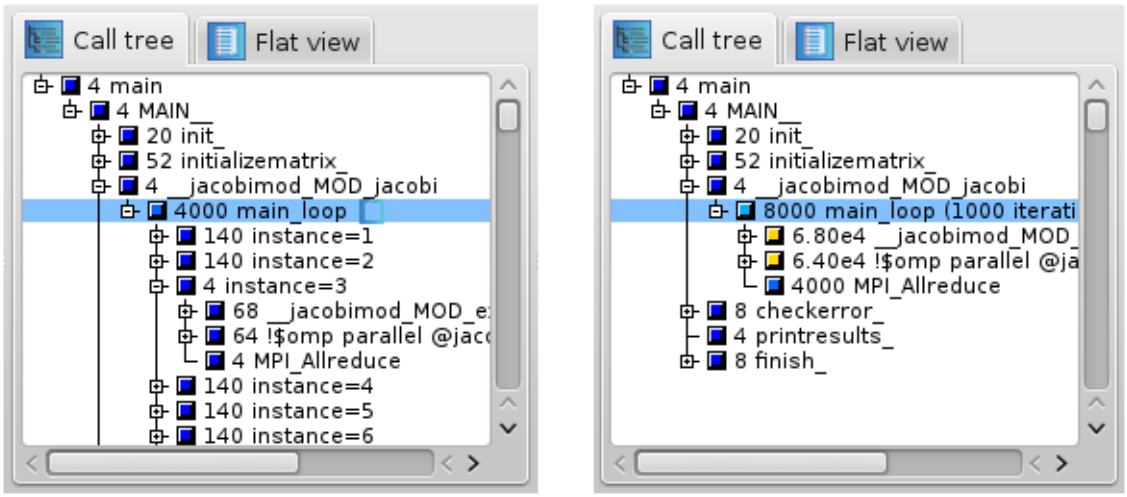

Figure 1.9: The item main_loop with 1000 iteration is marked as a loop. The aggregated view on the right is the result of selecting ”Hide iterations”.

begin and the end of the relevant source code region are highlighted. If the specified source file is not found, you are asked to chose a file to open.

c) **Set as loop:** Marks the selected tree item as loop. All subitems are treated as iterations. An additional context menu item "Hide iterations" appears.

22. **Called region:** For call trees only. Enabled only if there is a reference node. Offers information about the reference node.

    a) **Info:** Gives some short information about the reference node.

    b) **Online description:** Shows some (usually more extensive) online description for the reference node. Disabled if no online description is available.

    c) **Location:** Displays information about the module and position within the module (line numbers) where the callee method of the reference node is defined.

    d) **Source code:** Opens an editor for displaying, editing, and saving the source code of the callee of the reference node. Begin and end of the relevant region are highlighted. If the specified source code does not exists, you are asked to choose a file to open.

23. **Min/max values:** Not for metric trees. Here you can activate and deactivate the application of user-defined minimal and maximal values for the color extremes, i.e., the values corresponding to the left and right end of the color legend. If you activate user-defined values for the color extremes, you are asked to define two values that should correspond to the minimal and to the maximal colors. All values outside of this interval will get the color gray. Note that canceling any of the input windows causes no changes in the coloring method. If user-defined min/max

values are activated, the selected value information widget (see Section1.5.2.5) displays a "(u)" for "user-defined" behind the minimal and maximal color values.

24. **Statistics:** Only available if a statistics file for the current CUBE file is provided. Displays statistical information about the instances of the selected metric in the form of a box plot. For an in-depth explanation of this feature see subsection1.6.6.1.

25. **Max severity in trace browser:** Only available for metric and call trees and only if a statistics file providing information about the most severe instance(s) of the selected metric is present. If CUBE is already connected to a trace browser (via *File* ⇒ *Connect to trace browser*), the timeline display of the trace browser is zoomed to the position of the occurrence of the most severe pattern so that the cause for the pattern can be examined further. For a more detailed explanation of this feature see subsection1.6.6.2.

26. **Cut all tree:** For call trees only. Enabled only if clicked over item in call tree. Offers different modification possibilities:

    a) **Set as root:** Removes all call path above the selected item and sets selected call path as a root node.

    b) **Prune element:** Removes the selected item and all its children. Its inclusive value will be added then to the exclusive value of its parent.

    c) **Set as leaf:** Removes all children of its element and add their inclusive values to its exclusive value.

27. **Create derived metric (as root or a child)** For metric tree only. It offers a dialog1.10 to create a new derived metric as a root metric if clicked over an empty part of window or selected via submenu **as a root**. An it creates the new metric as a child metric if clicked over another metric or selected via submenu **as a child**.

    Documentation about derived metrics see in[12]

    Some details about the fields in the dialog:

    a) **Select metric from collection:** Provides a list of predefined derived metric, which might be helpful for the analysis.

    b) **Derived metric type:** Selects the type of the derived metrics. Available are : `Postderived metric`, `Prederived exclusive metric` and `Prederived inclusive metric`.

    c) **Display name:** Sets the display name of the metric in the metric tree.

    d) **Unique name:** Sets the unique name of the metric. There is no check done if another metric is present with the same unique name.

    e) **Data type :** For derived metrics it is preselected and is always `DOUBLE`.

    f) **Unit of measurement:** Selects a unit of measurement. It is a user defined string.

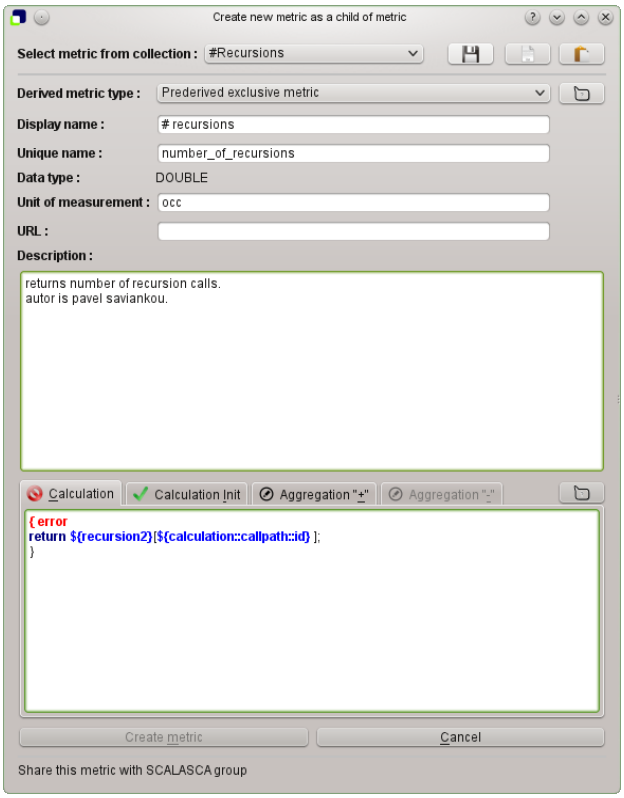

Figure 1.10: Create derived metric

g) **URL:** Selects a URL with the documentation about this metric.

h) **Description:** Describes a metric.

i) **Calculation:** Field where one enters the CubePL expression for the derived metric. Automatic syntax check is done. If there is a syntax error, dialog highlights the place of the error and gives an error message.

j) **Calculation Init:** Field where one enters the initialisation CubePL expression for the derived metric,which is executed only once after metric creation.

   Automatic syntax check is done. If there is a syntax error, dialog highlights the place of the error and gives an error message.

k) **Aggregaton "+":**Prederived metrics can specify an expression for the operator "+" in the aggregation formula. In this field one can redefine it.

   Automatic syntax check is done. If there is a syntax error, dialog highlights the place of the error and gives an error message.

l) **Calculation "-":** Prederived inclusive metric can specify an expression for the operator "-" in the aggregation formula. In this field one can redefine it.

   Automatic syntax check is done. If there is a syntax error, dialog highlights

the place of the error and gives an error message.

m) **Create metric** - closes dialog and creates metric with parameters, set in this dialog. Enabled if syntax is OK type of metric is selected and fields `Unique name` `Display name` are set.

n) **Cancel** - closes dialog without creating any metric.

o) **Share this metric with SCALASCA group** - Offers you to sent the metric definition via email to the SCALASCA group, so it might be included into the library of derived metrics in the future releases. Enabled only if definition of metric is valid.

To simplify the creation of a derived metric a little bit there is a way to fill the fields of this dialog automatically.

If one prepares a file with the following syntax one can select it and open "drop" on dialog via drag'n'drop, or copy its content into clipboard and paste in the dialog.

Example of a syntax of this `file:`

```
metric type: postderived
display name: Average execution time
unique name: kenobi
uom:sec
url: https://scalasca.org/documentation.html#kenobi
description:Calculates an average execution time
#
# Here is the Kenobi metric
#
cubepl expression: metric::time(i)/metric::visits(e)

cubepl init expression:

cubepl plus expression:  arg1 + arg2

cubepl minus expression: arg1 - arg2
```

`metric type` can have values: `postderived`, `prederived_exclusive` or `prederived_inclusive`.

28. **Remove metric** For metric tree only. Removes metric from the metric tree.
29. **Edit metric** For metric tree only. It offers a dialog to edit expressions (standard, initialisation, aggregation) of a derived metric. Enabled if clicket metric is a derived metric. Window for editing is same like in "Create derived metric" case.
30. **Sort by value (descending):** For flat call profiles only. Sorts the nodes by their current values in descending order. Note that if an item is expanded its exclusive value is taken for sorting, otherwise its inclusive value.
31. **Sort by name (ascending):** For flat call profiles only. Sorts the nodes alphabeti-

cally by name in ascending order.

1.5.2.5 Selected value info

Below each pane there is a selected value information widget. If no data is loaded, the widget is empty. Otherwise, the widget displays more extensive and precise information about the selected values in the tree above. This information widget and the topologies may have different precision settings than the trees, such that there is the possibility to display more precise information here than in the trees (see Section1.6.4.2, menu *Display* ⇒ *Precision*).

The widget has a 3-line display. The first line displays at most 4 numbers. The left-most number shows the smallest value in the tree (or 0.0 in any percentage value mode for trees, or the user-defined minimal value for coloring if activated), and the right-most number shows the largest value in the tree (or 100.0 in any percentage value mode in trees, or the user-defined maximal value for coloring if activated). Between these two numbers the current value of the selected node is displayed, if it is defined. Additionally, in the absolute value mode it is followed by the percentage of the selected value on the scale between the minimal and maximal values, shown in brackets. Note that the values of expanded non-leaf system nodes and of nodes of trees on the left-hand side of the metric tree are not defined. If the value mode is not the absolute value mode, then in the second line similar information is displayed for the absolute values in a light gray color.

In case of multiple selection, the information refers to the sum of all selected values. In case of multiple selection in system trees in the peer distribution and in the peer percent modes, this sum does not state any valuable information, but is displayed for consistency reasons.

If the widget width is not large enough to display all numbers in the given precision, then a part of the number displays get cut down and a " ... " indicates that not all digits could be displayed.

Below these numbers, in the third line, a small color bar shows the position of the color of the selected node in the color legend. In case of undefined values, the legend is filled with a gray grid.

1.5.2.6 Color legend

By default, the colors are taken from a spectrum ranging from blue over cyan, green, and yellow to red, representing the whole range of possible values. You can change the color settings in the menu,`Display ⇒ General coloring`, see Section1.6.4.2. Exact zero values are represented by the color white (in topologies you can decide whether you would like to use white or the minimal color, see Section1.6.4.2, menu `Topology`).

1.5.2.7 Status Bar

The status bar displays some status information, like state of execution for longer procedures, hints for menus the mouse pointing at etc.

The status bar shows the most recent log message. By clicking on it the complete log becomes visible.

1.6 Plugins

The features of cube can be extended using plugins. There is a set of predefined plugins which are described in the following sections. Before a cube file is loaded, the Plugin menu only contains the menu item "Initial activation settings". Selecting this item shows the following dialog, which shows all available plugins.

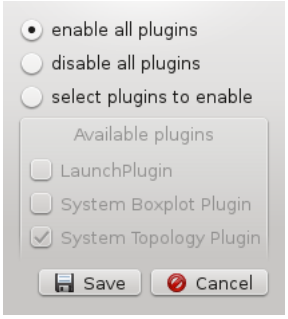

Figure 1.11: plugin settings dialog

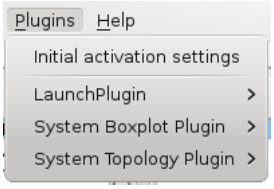

Figure 1.12: plugin menu

You may enable or disable all plugins, or select individual plugins that will be activated or deactivated. After loading a cube file, all suitable plugins are activated. Each plugin adds a submenu (see Figure1.13) to the Plugins menu, which allows to enable or disable it. It may also add further menu items.

1.6.1 topology

In many parallel applications, each process (or thread) communicates only with a limited number of processes. The parallel algorithm divides the application domain into smaller chunks known as sub-domains. A process usually communicates with processes owning sub-domains adjacent to its own. The mapping of data onto processes and the neighborhood relationship resulting from this mapping is called *virtual topology*. Many applications use one or more virtual topologies specified as multi-dimensional Cartesian grids.

Another sort of topologies are *physical topologies* reflecting the hardware structure on which the application was run. A typical three-dimensional physical topology is given by the (hardware) nodes in the first dimension, and the arrangement of cores/processors on nodes in further two dimensions.

The CUBE display supports multi-dimensional Cartesian grids, where grids with high dimensionality can be sliced or folded down to two or three dimensions for presentation. If the currently opened cube file defines one or more such topologies, separate tabs are available for each using the topology name when one is provided. The topology display shows performance data mapped onto the Cartesian topology of the application. The corresponding grid is specified by the number of dimensions and the size of each dimension. Threads/processes are attached to the grid elements, as specified by the CUBE file. Not all system items have to be attached to a grid element, and not every grid element has a system item attached. An example of a three-dimensional topology is shown on Figure1.13 . Note that the topology toolbar is enabled when a topology is available to be displayed.

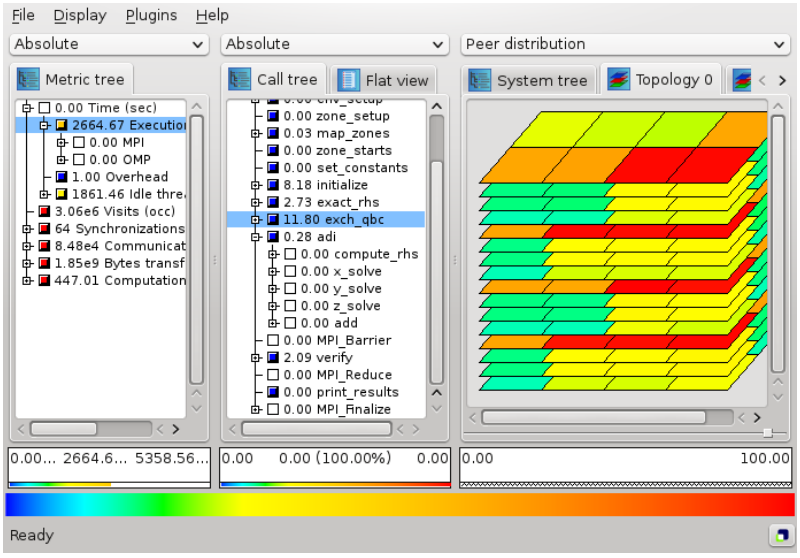

Figure 1.13: Topology Displays

The Cartesian grid is presented by planes stacked on top of each other in a three dimensional projection. The number of planes depends on the number of dimensions in the grid. Each plane is divided into tiles (typically shown as rombi). The number of tiles depends on the dimension size. Each tile represents a system resource (e.g., a process) of the application and has a coordinate associated with it.

The current value of each grid element (with respect to the selections on the left-hand side and to the current value mode) is represented by coloring the grid element. Coloring is based on a value scale from 0.0 to 100.0. Grid elements without having a system item attached to it are colored gray. See Section1.6.4.2 (menu *Topology*) for further topology-specific coloring settings. For example, the upper topology in Figure1.13 is drawn wit black lines, the 2D topology in Figure1.14 is drawn without lines.

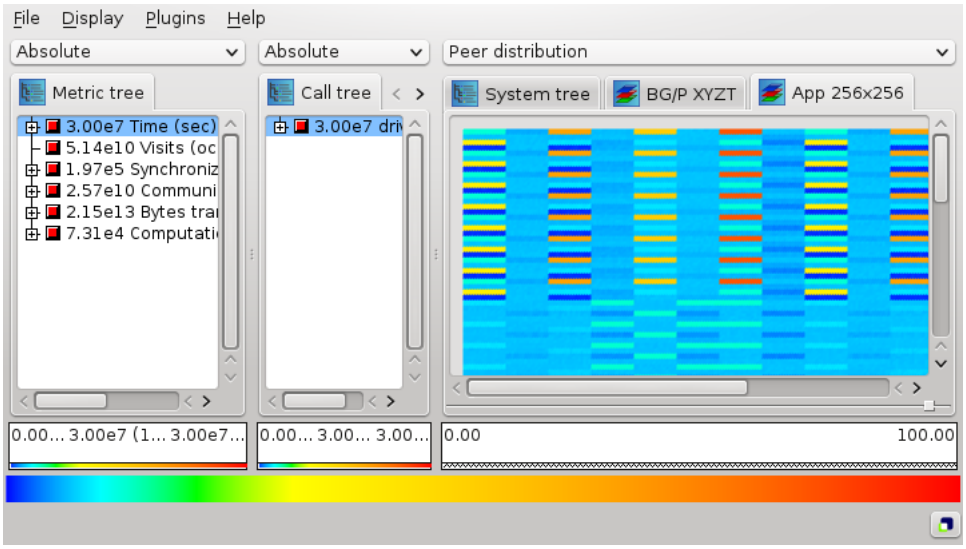

Figure 1.14: Topology Displays

If the selected system item occurs in the topology, it is marked by an additional frame and by additional lines at the side of the plane which contains the corresponding grid point, such that the selected item's position is also visible if the corresponding plane is not completely visible.

If zooming into planes is enabled, the plane containing the recently selected item is selected and the plane distance is adjusted to show this plane complely.

Selecting a collapsed tree in the system tree selects all its children in the topology view.

Besides the functions offered by the topology toolbar (see1.6.3.2), the following functionality is supported:

1. **Item selection:** You can change the current system selection by left-clicking on a grid element which has a system item assigned to it (resulting in the selection of

that system item). Multiple items may be selected or deselected by holding down the Ctrl key while clicking on an item.

2. **Info:** By right-clicking on a grid element, an information widget appears with information about the system item assigned to it. The information contains
   - the coordinate of the grid point in each topology dimension,
   - the hardware node to which the attached system item belongs to,
   - the system item's name,
   - its MPI rank,
   - its identifier,
   - and its value, followed by the percentage of this value on the scale between the minimal and maximal topology values.
3. **Rotation about the x and y axes:** can be done with left-mouse drag (click and hold the left-mouse button while moving the mouse).
4. **Increasing/decreasing the distance between the planes:** with Ctrl+<left-mouse drag>
5. **Moving the whole topology up/down/left/right:** with Shift+<left-mouse drag>

1.6.1.1 Topology mapping panel

If the number of topology dimensions is larger than three, the first three dimensions are shown and an additional control panel appears below the displayed topology. This panel allows rearranging topology dimensions on the x, y and z axes, as well as slicing or folding of higher dimensionality topologies for presentation in three or fewer dimensions.

Rearranging topology dimensions is achieved simply by dragging the topology dimension labels to the desired axis. When dragged on top of an existing topology dimension label, the two are exchanged.

When slicing, select up to three of the dimensions to display completely and choose one element of each of the remaining dimensions. The example in Figure1.15 shows a topology with 4 dimensions (32x16x32x4) labelled X, Y, Z and T. The first element of the 4th dimension (T) is automatically selected. By clicking on the button above the T, an index in this dimension from 0 to 3 can be chosen. If the index is set to *all*, the selection becomes invalid until an index of another dimension is selected.

Alternatively, the folding mode can be activated by clicking on the *fold* button. This mode is available for topologies with four to six dimensions and allows to display all elements by folding two dimensions into one. Every dimension appears in a box, with can be dragged into one of the three container boxes for the displayed dimensions x, y and z. In folding mode, the color of the inner borders is changed into gray. The black bordered rectangles show the element borders of each of the three displayed dimensions.

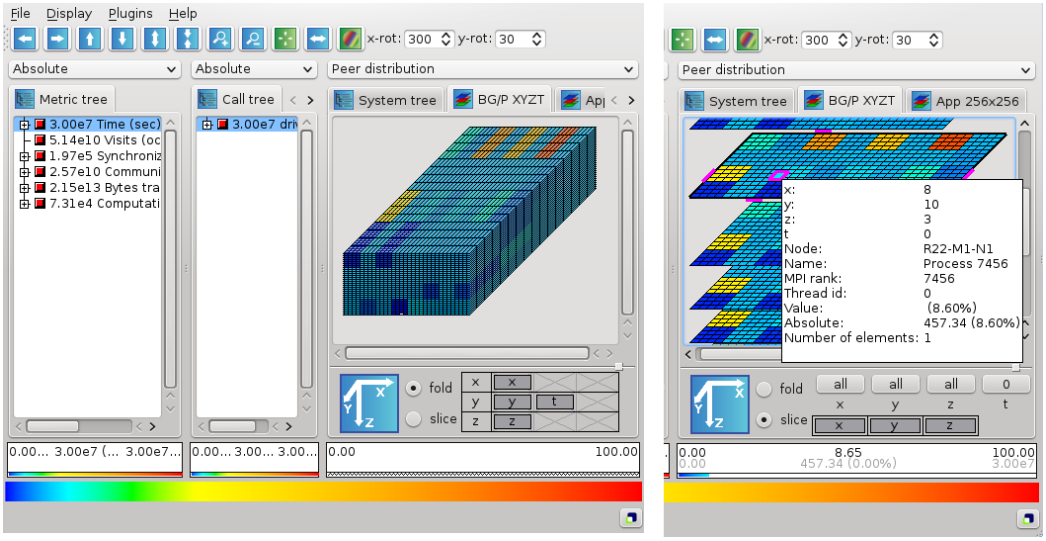

Figure 1.15: 4-dimensional example

The right image in Figure1.15 shows the folding of dimension Z with dimension T. One element with index (0,0,1,3) has been selected by clicking with the right mouse button into it. All elements inside the black rectancle around the selection belong to Z index one. The gray lines devide the rectangle into four elements which correspond to the elements of dimension T with index 0 to 3.

1.6.2 topology

Topology: The topology menu offers the following functions related to the topology display described in Section1.6.2:

1. **Item coloring:** Offers a choice how zero-valued system nodes should be colored in the topology display. The two offered options are either to use white or to use white only if all system leaf values are zero and use the minimal color otherwise.
2. **Line coloring:** Allows to define the color of the lines in topology painting. Available colors are black, gray, white, or no lines.
3. **Toolbar:** This menu item allows to specify if the topology toolbar buttons should be labeled by icons, by a text description, or if the toolbar should be hidden. For more information about the toolbar see Section1.6.3.2.
4. **Show also unused hardware in topology:** If not checked, unused topology planes, i.e., planes whose grid elements don't have any processes/threads assigned to, are hidden. Unused plane elements, if not hidden, are colored gray.
5. **Topology antialiasing:** If checked, anti-aliasing is used when drawing lines in the

topologies.

6. **Zoom into current plane:** If checked, the plane containing the recently selected item is shown completely. It is never covered by a neighbor plane.

1.6.2.1 toolbar

The system pane may contain topology displays if corresponding data is specified in the CUBE file. Basically, a topology display draws a two- or three-dimensional grid, in the form of some planes placed one above the other. Each plane consists of a two-dimensional grid of processes or threads.

The toolbar is enabled only if the system pane shows a topology display, and it offers functions to manipulate the display of the above grid planes. The toolbar can be labeled by icons, by text, or it can be hidden, see menu *Topology* ⇒ *Toolbar* in Section1.6.4.2. The toolbar buttons have tool tips, i.e., a short description pops up if the toolbar is enabled and you move the mouse above a button.

The functions are the following, listed from the left to the right in the topology toolbar:

Move left Moves the whole topology to the left.

Move right Moves the whole topology to the right.

Move up Moves the whole topology upwards.

Move down Moves the whole topology downwards.

Increase plane distance Increase the distance between the planes of the topology.

Decrease plane distance Decrease the distance between the planes of the topology.

Zoom in Enlarge the topology.

Zoom out Scale down the topology.

Reset Reset the display. It scales the topology such that it fits into the visible rectangle, and transforms it into a default position.

Scale into window It scales the topology such that it fits into the visible rectangle, without transformations.

Set minimum/maximum values for coloring Similarly to the functions offered in the context menu of trees (see Section1.5.2.4), you can activate and deactivate the application of user-defined minimal and maximal values for the color extremes, i.e., the values corresponding to the left and right end of the color legend. If you activate user-defined values for the color extremes, you are asked to define two values that should correspond to the minimal and to the maximal colors. All values outside of this interval will get the color gray. Note that canceling any of the input

windows causes no changes in the coloring method. If user-defined min/max values are activated, the selected value information widget displays a “(u)” for“user-defined” behind the minimal and maximal color values.

x-rotation Rotate the topology cube about the x-axis with the defined angle.

y-rotation Rotate the topology cube about the y-axis with the defined angle.

Dimension order for the topology displays This button no longer exists, but formerly allowed the order of topology dimensions to be adjusted: this is now done with the control panel at the bottom of the topology pane.

Using the grip at the left of the toolbar, it can be dragged to another position or detached entirely from the main window. The toolbar can also be closed after a right-click in the grip.

1.6.3 BarplotPlugin

BARPLOT plugin is a CUBE plugin that plots vertical bar graph for the CUBE file which has iterations. Horizontal axis shows different iterations being compared and on vertical axis, several operations can be used to represent the value. The User can apply different metrics and call paths on the bar graph.

1.6.3.1 Basic Principles

As a start point, it should be mentioned that BARPLOT works only on a CUBE file that has iterations. For those files which have not, user would face the warning on the terminal : `"No iterations for Barplot"` and the plugin will not be shown.

By loading the plugin, on *system dimension*, the corresponding tab, *Barplot*, will be added. In the *Barplot* tab, the user can select different operations and assign desired color to them. Figure1.16 displays a view of it.

User can select different metrics such as *Visits* and *Time*, by clicking on them in *metric dimension*. In addition, it is possible to get a BARPLOT for different *call paths* of iterations, via clicking on them. However, for *call paths* that are not located in iterations, like *input_in* in figure1.17 , no bar graph is displayed and user face the message `"No data to display"` on the window.

Furthermore, the values on BARPLOT, can be evaluated in *Inclusive* and *Exclusive* manner. Therefore, user can easily collapse the tree on *call path* and click on the desired path to get the exclusive value of it.

Additionally, the exact calculated values can be seen by clicking left button of mouse on the desired position on the graph, a tooltip would display a value corresponding to the iteration.

In a situation that user needs to store the graph, it is just needed to do right click on a

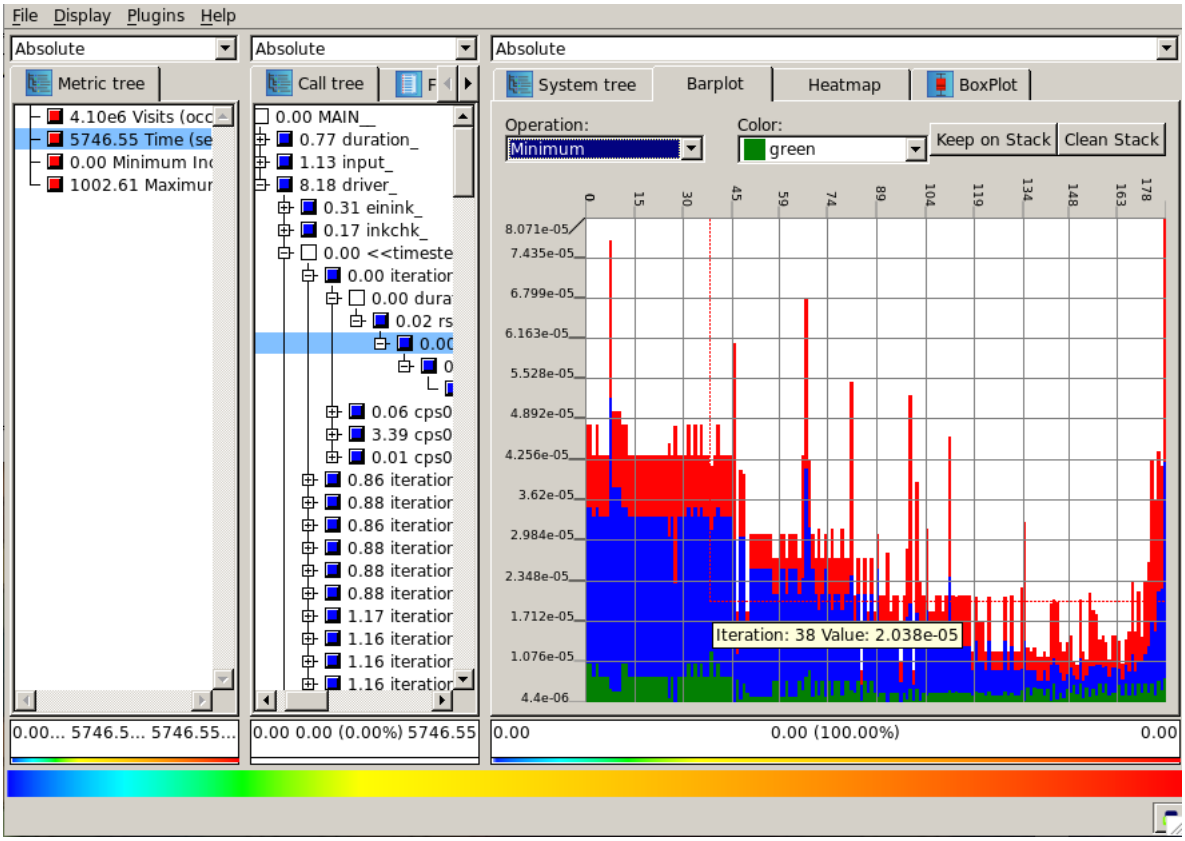

Figure 1.16: BARPLOT display window

graph, and select "`Save as image`", then the *Save dialog* will be opened to specifying the path and name of the PNG file.

1.6.3.2 Toolbar

On the top of the Barplot space, there is a toolbar that allows user to specify the kind of an operation and its color(Figure1.18).

By *operation* item, the user can select different operations, *Minimum*, *Maximum*, *Average*, *Median*, *1st Quartile* and *3rd Quartile* or the combination of *Maximum*, *Minimum* and *Average*. This provides the situation for the user to have different values for comparing at one time. These operations are done on all threads in each iterations. For instance, by *Minimum* operation, the minimum value among the existing threads for each iteration, is calculated and plotted. They are kind of statistical measurements.

Color item offers a color for an *operation*, however for each *operation*, a default color is assigned automatically. By changing the *operation*, corresponding color will be shown on *color* combo box. In a situation that different bar graphs are overlaid on each other,

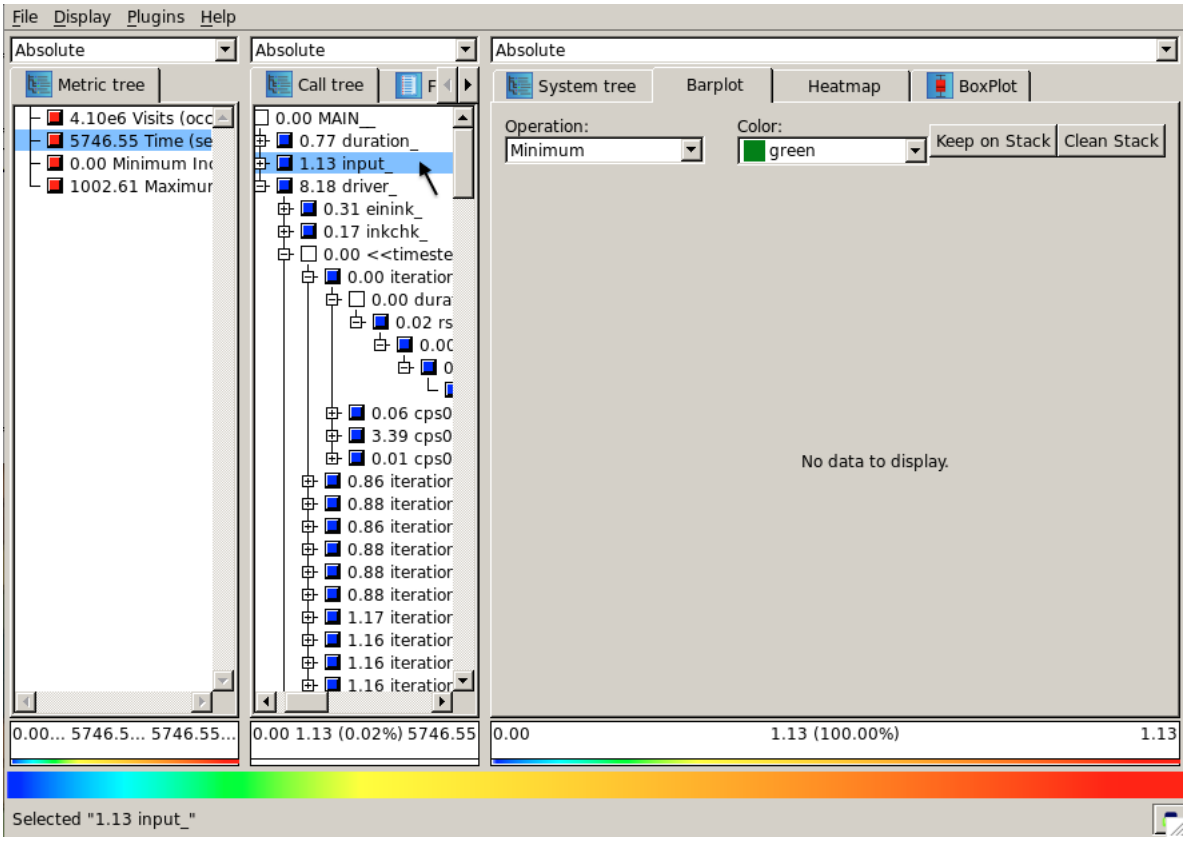

Figure 1.17: No data to display

each graph will be shown by different color in order to distinguish various graphs.

In addition to above items, two buttons are also designed to manage the order of the bar graphs.

Keep on Stack: It is possible that user intents to compare different graphs by laying them on each other. For this matter, a push-button *keep on stack* is defined. Generally, by clicking on each *call path* or *metric*, a responding graph is replaced the previous one in the stack. In a situation, that the user intends to compare the next graph by the existing one, at one time, it is needed to click on the button *keep on the stack*, then the next graph will be added over the previous one, or in another words, it is overlaid on the last graph. If its values are less than the previous graph, user can see two graphs by different colors that help him/her in comparing, and in a situation that new values are greater than previous one, the new one will cover the previous with fresh color. Therefore, for keeping the top row of the stack, the user should click on the *keep the stack* button, otherwise the coming values will replace the last one.

Clean Stack: By clicking this button, all displayed graphs, are erased and the stack will be empty.

Figure 1.18: BARPLOT toolbar

1.6.3.3 Menu Bar

Plugin menu offers the general function to enable or disable a plugin, and specific functions for each plugin. *Barplot plugin* provides the following functions in two areas, Measurement Customization and Threads Ruler Customization(Fiqure1.22).

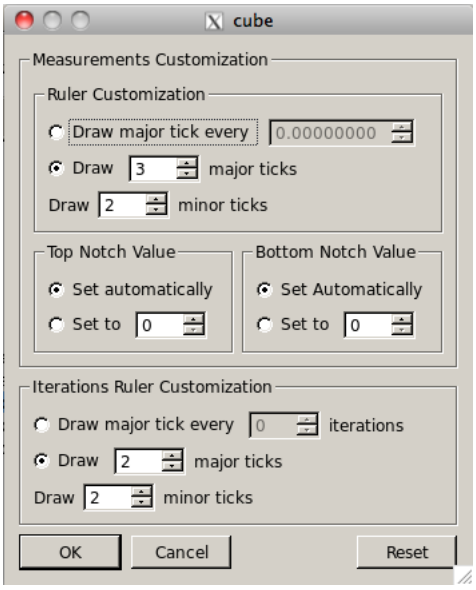

Figure 1.19: BARPLOT menu

Ruler Customization: User can modify the number of major and minor ticks of the ruler on vertical axis. For adjusting the major vertical ticks, user can set the drawing intervals or the number of ticks. By specifying the number of major ticks, the length of the vertical axis will be divided to the specified number and major ticks are drawn by length longer than minor ticks. Then in each divided length, if there is enough space, the specified number of minor ticks will be displayed. It is possible that the user set major ticks by interval. In order to do that, select the major ticks by interval option, and set the interval value. Therefore, after each interval, one major tick will be drawn.

Top Notch Value: The value of the top notch on a vertical axis can be altered by user as well as automatically. Therefore, due to scale issue, it can affect on the drawing of the graph.

Button Notch Value: The value of the button notch on a vertical axis can be altered by user as well as automatically. Therefore, due to scale issue,it can affect on the drawing of the graph.

Iterations Ruler Customization: User can modify the number of major and minor ticks of the ruler on horizontal axis. For adjusting the major horizontal ticks, user can set the drawing intervals or the number of ticks. By specifying the number of major ticks, the width of the horizontal axis will be divided to the specified number and major ticks are drawn by length longer than minor ticks. Then in each divided length, if there is enough space, the specified number of minor ticks will be displayed. It is possible that the user set major ticks by interval of iterations. In order to do that, select the major ticks by interval option, and set the interval. Therefore, after each specified number of iterations, one major tick will be drawn.

1.6.4 HeatmapPlugin

HEATMAP plugin is a CUBE plugin that represents the value of the thread in each iteration, as colors. The User can apply different metrics and call paths on heatmap graph.

1.6.4.1 Basic Principles

As a start point, it should be mentioned that HEATMAP works only on CUBE file that has iterations. For those files which have not, user would face the warning on the terminal : `"No iterations for Heatmap"` and the plugin will not be shown.

By loading the plugin, on *system dimension*, the corresponding tab, *Heatmap*, will be added. Figure1.20 displays a view of it.

User can select different metrics such as *Visits* and *Time*, by clicking on them in *metric dimension*. In addition, it is possible to get a HEATMAP for different *call paths* of iterations, via clicking on them. However, for *call paths* that are not located in iterations, like *input_in* figure1.21 , no heatmap graph is displayed and user face the message `"No data to display"` on a window.

Furthermore, the values on HEATMAP, can be evaluated in *Inclusive* and *Exclusive* manner. Therefore, user can easily collapse the tree on *call path* and click on the desired path to get the exclusive value of it.

Additionally, the exact calculated values can be seen by clicking left button of mouse on the desired position on the graph, a tooltip would display a value corresponding to the iteration.

In a situation that user needs to store the graph, it is just needed to do right click on a graph, and select `"Save as image"`, then the *Save dialog* will be opened to specifying the path and name of the PNG file.

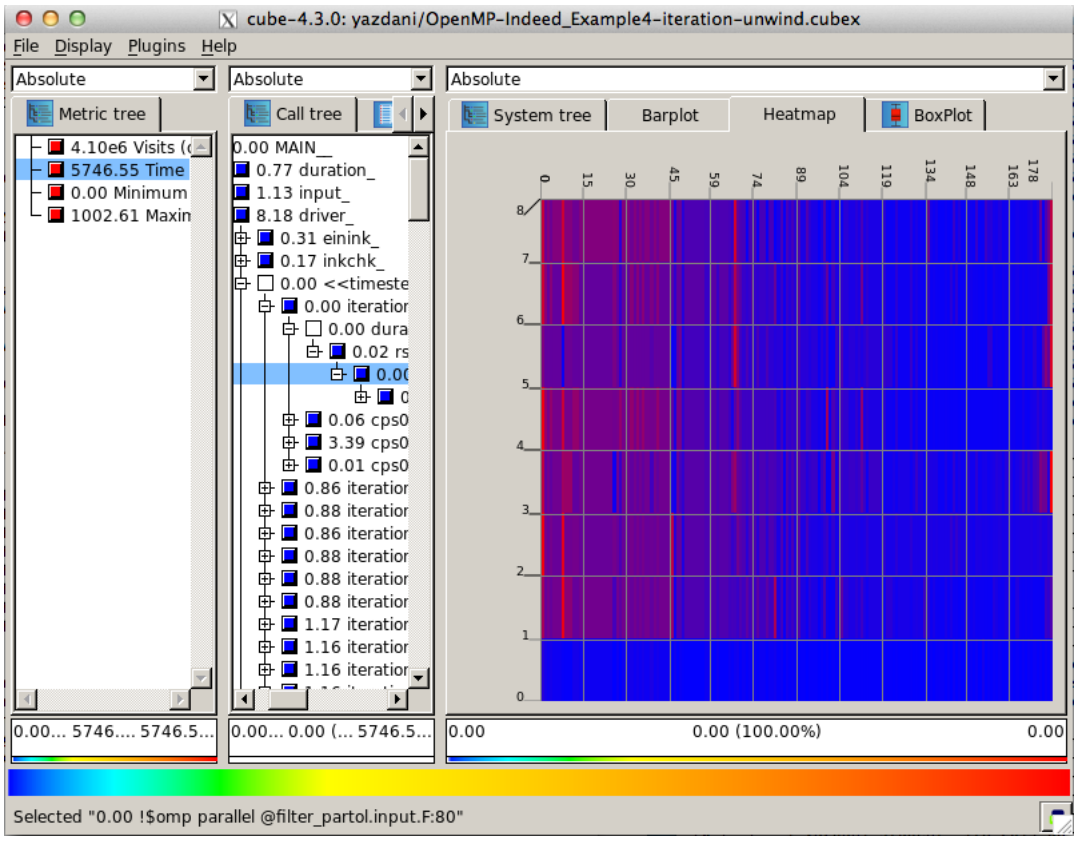

Figure 1.20: HEATMAP display window

1.6.4.2 Menu Heatmap

Plugin menu offers the general function to enable or disable a plugin, and specific functions for each plugin. *Heatmap plugin* provides the following functions in two areas, horizontal tick and vertical ticks(Fiqure1.22).

Horizontal ticks: For adjusting the major horizontal ticks, user can set the drawing intervals or the number of ticks. By specifying the number of major ticks, the width of the horizontal axis will be divided to the specified number and major ticks are drawn by length longer than minor ticks. Then in each divided length, if there is enough space, the specified number of minor ticks will be displayed.

Also, it is possible that the user set major ticks by interval of iterations. In order to do that, select the major ticks by interval option, and set the interval. Therefore, after each specified number of iterations, one major tick will be drawn.

Vertical ticks: For adjusting the major vertical ticks, user can set the drawing intervals or the number of ticks. By specifying the number of major ticks, the length of the vertical axis will be divided to the specified number and major ticks are drawn by length longer

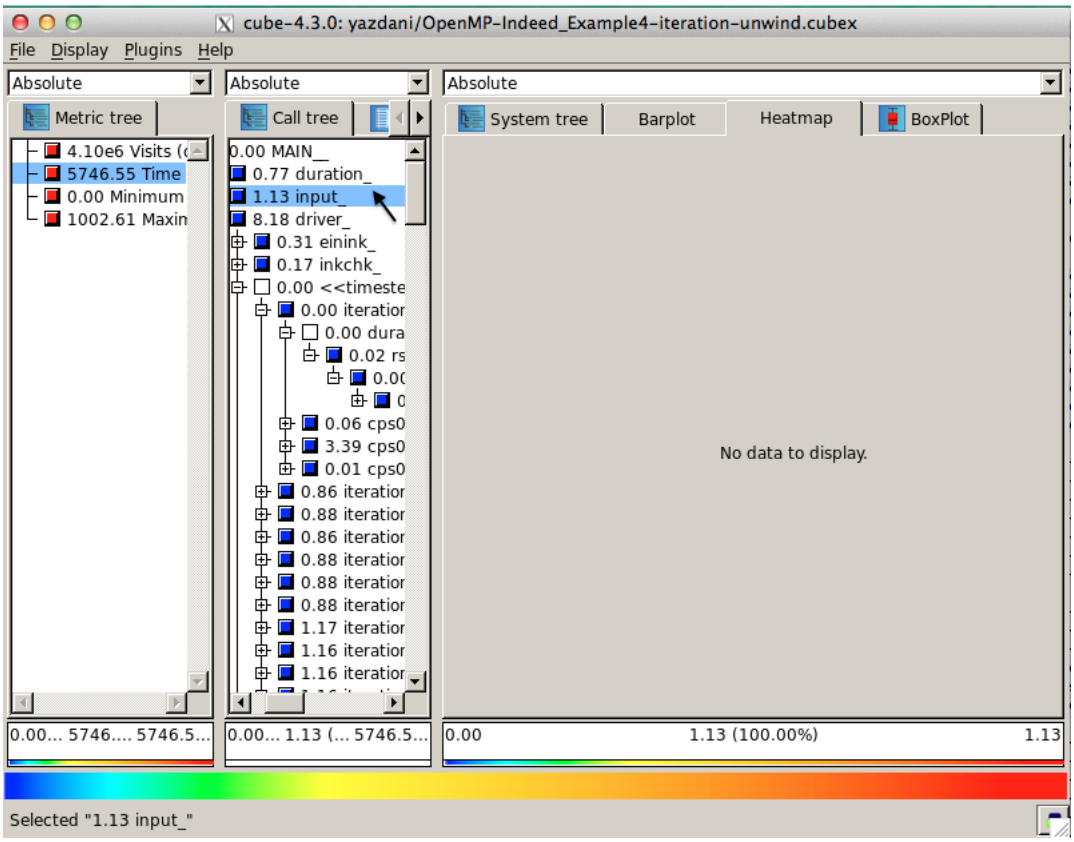

Figure 1.21: No data to display

than minor ticks. Then in each divided length, if there is enough space, the specified number of minor ticks will be displayed.

Also, it is possible that the user set major ticks by interval of threads. In order to do that, select the major ticks by interval option, and set the interval. Therefore, after each specified number of threads, one major tick will be drawn.

1.6.5 SystemBoxplotPlugin

This plugin adds a boxplot statistics display tab next to the system tree tab. It shows a box-and-whisker distribution of metric severity values for the currently active subset of system resources (typically threads). The active subset is changed via the combobox menu at the bottom of the pane, and the *y*-axis scale is adjusted via the display mode combobox at the top of the pane.

The vertical whisker ranges from the smallest value (minimum) and to the largest value (maximum), while the bottom and top of the box mark the lower quartile (Q1) and upper

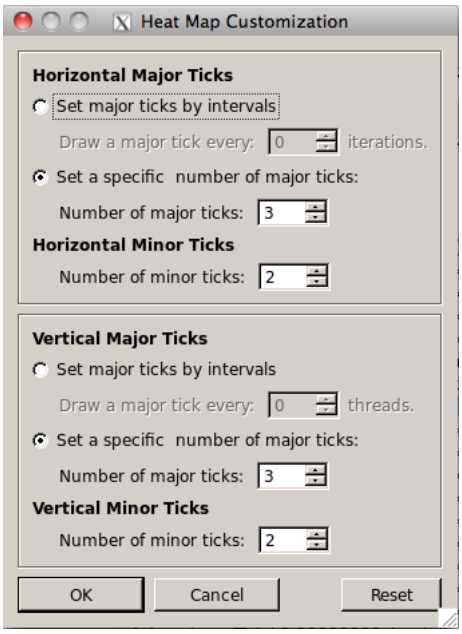
Heat Map Customization
Horizontal Major Ticks
Set major ticks by intervals
Draw a major tick every: 0 iterations.
Set a specific number of major ticks:
Number of major ticks: 3
Horizontal Minor Ticks
Number of minor ticks: 2
Vertical Major Ticks
Set major ticks by intervals
Draw a major tick every: 0 threads.
Set a specific number of major ticks:
Number of major ticks: 3
Vertical Minor Ticks
Number of minor ticks: 2
OK
Cancel
Reset

Figure 1.22: HEATMAP menu

quartile (Q3). Within the box, the bold horizontal line represents the median (Q2) and the dashed line the mean value.

To see the statistics as numeric values in a separate window, use <left-mouse click> inside the chart. Zooming into the boxplot is done with <left-mouse drag> from top to bottom, and reset with a <middle-mouse click> inside the chart.

%deprecated

1.6.6 Features enabled through statistic files

In this section we will explain two features -- namely the display of statistical information about performance patterns which represent performance problems and the display of the most severe instances of these patterns in a trace browser -- which both are only available if a statistic file for the currently opened CUBE file is present. Currently, such a statistic file can be generated by the SCOUT analyzer[5]. The file format of statistic files is described in the Appendix 3.1.

For CUBE to recognize the statistic file, it must be placed in the same directory as the CUBE file. The basename of the statistic file should be identical to that of the CUBE file, but with the suffix `.stat`. For example, when the CUBE file is called `trace.cubex`, the corresponding statistic file is called `trace.stat`.

1.6.6.1 Statistical information about performance patterns

If a statistic file is provided, you can view statistical information about one or multiple patterns (for example in order to compare them). This is done by selecting the desired metrics in the metric tree and then selecting the *Statistics* menu item in the context menu. This brings up the box plot window as shown in Figure1.23 .

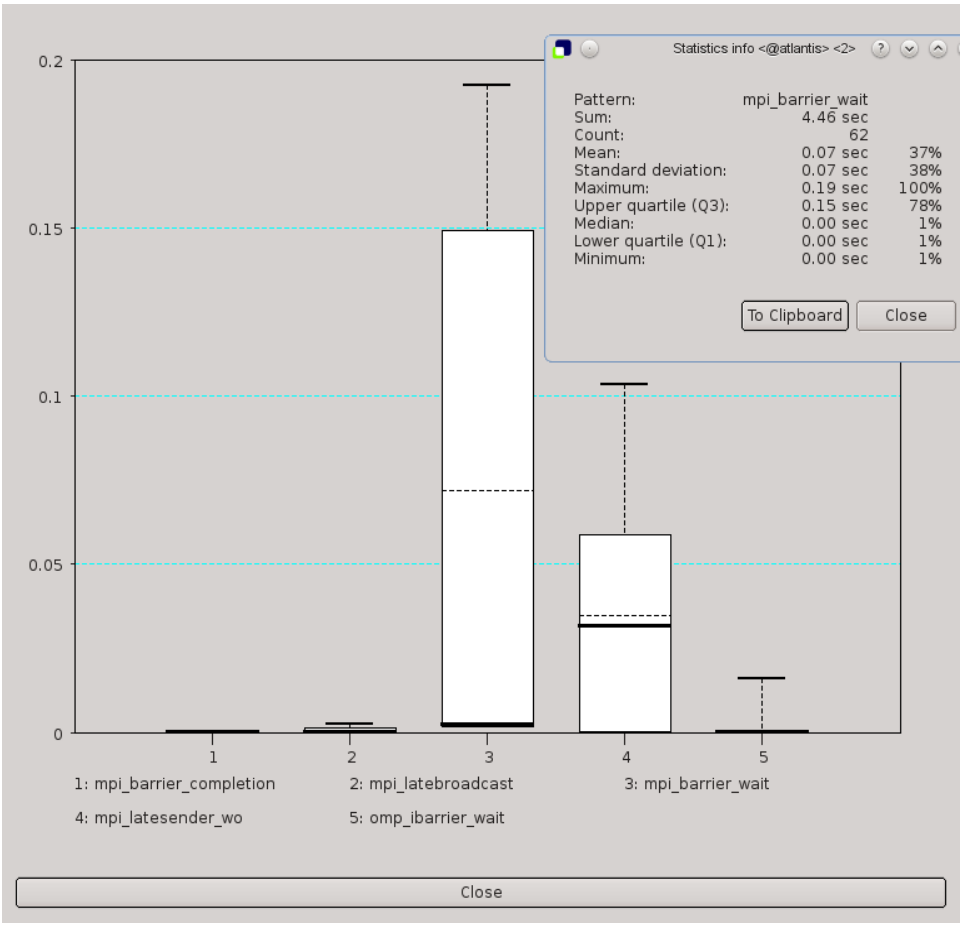

Figure 1.23: Screenshot of a box plot as shown by CUBE displaying statistical information about the selected patterns. The additional window on the top right displaying the exact values of the statistics.

The box plot shows a graphical representation of the statistical data of the selected patterns. The slender black lines on the top and the bottom designate the maximum and the minimum measured severity of the pattern, respectively. The lower and the upper borders of the white box indicate the values of the 25% and 75% quantile. The thick line inside the box represents the median of the values, while the dashed line indicates the mean.

There are two ways of interacting with the box plot. You can zoom to a certain interval

on the y-axis by clicking on a position with the height of the desired maximal or minimal value and by consecutively dragging the mouse to a position with the height of the corresponding other extreme value. You can reset the view (i.e., to undo all zooming) by clicking the middle mouse button somewhere on the box plot.

If you are interested in more precise values for the severity statistics of a certain metric, you can click somewhere in the column of the desired metric, which will yield a small window (as shown in the top right corner of Figure1.23) displaying the exact values of the statistics.

1.6.6.2 Display of most severe pattern instances using a trace browser

If a statistic file also contains information about the most severe instances of certain patterns, CUBE can be connected to a trace browser (currently Vampir[8, 9] and Paraver [6, 7] are supported) in order to view the state of the program being analyzed at the time this most severe pattern instance occurred. For collective operations, the most severe instance is the one with the largest *sum* of the waiting times of all processes, which is not necessarily the one with the largest maximal waiting time of each individual process.

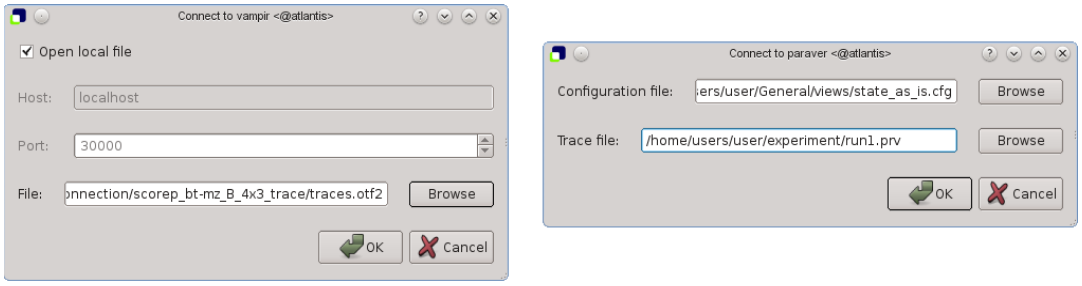

Figure 1.24: The dialog windows for a connection to Vampir and to Paraver

To use this feature, you first have to connect to a trace browser by using the *Connect to trace browser* menu item of the *File* menu, which offers to connect to Vampir as well as to Paraver. This will open one of the two dialog windows shown below.

For Vampir, you have to specify the host name and port of the Vampir server you want to connect to and the path of the trace file you want to load. This will launch the Vampir client (if it is correctly configured) and load the specified trace file. To configure Vampir so that it can be started automatically by CUBE, a service file `com.gwt.vampir.service`, describing the path to your Vampir client executable must be placed under `(/usr/share/dbus-1/service)` or `${HOME}/.local/share/dbus-1/services`. This service file must be exactly as shown below, with the exception that `Exec` should point to your Vampir client executable.

```
[D-BUS Service]
```

```
Name=com.gwt.vampir
Exec=/private/utils/bin/vng
```

An example of the `com.gwt.vampir.service` file

For Paraver, you have to specify a configuration file (which is used to initialize the Paraver window which is opened when zooming) as well as the path of the desired trace file. This will launch Paraver which will directly open the correct trace file. In order for CUBE to be able to launch Paraver, the executable directory of Paraver must be in your path.

It is also possible to connect to multiple trace browsers so that you can view a trace file in Paraver and Vampir simultaneously, but due to limitations with the Vampir client you can only have two Vampir clients running at the same time. All trace browsers will be zoomed simultaneously if you select a zoom command (as described below).

Once CUBE is connected to a trace browser you can select the *Max severity in trace browser* menu item of the metric tree so that all connected trace browsers are zoomed to the (globally) most severe instance of the selected pattern.

A more sophisticated feature of CUBE is the ability to zoom to the most severe instance of a pattern in a selected call path. This can be done by selecting a metric in the metric tree which will highlight the most severe call paths in the call tree. You can then use the context menu of the call tree to select the *Max severity in trace browser* menu item which will then zoom all connected trace browsers to the most severe instance of the selected pattern with respect to the chosen call path (see Figure1.25).

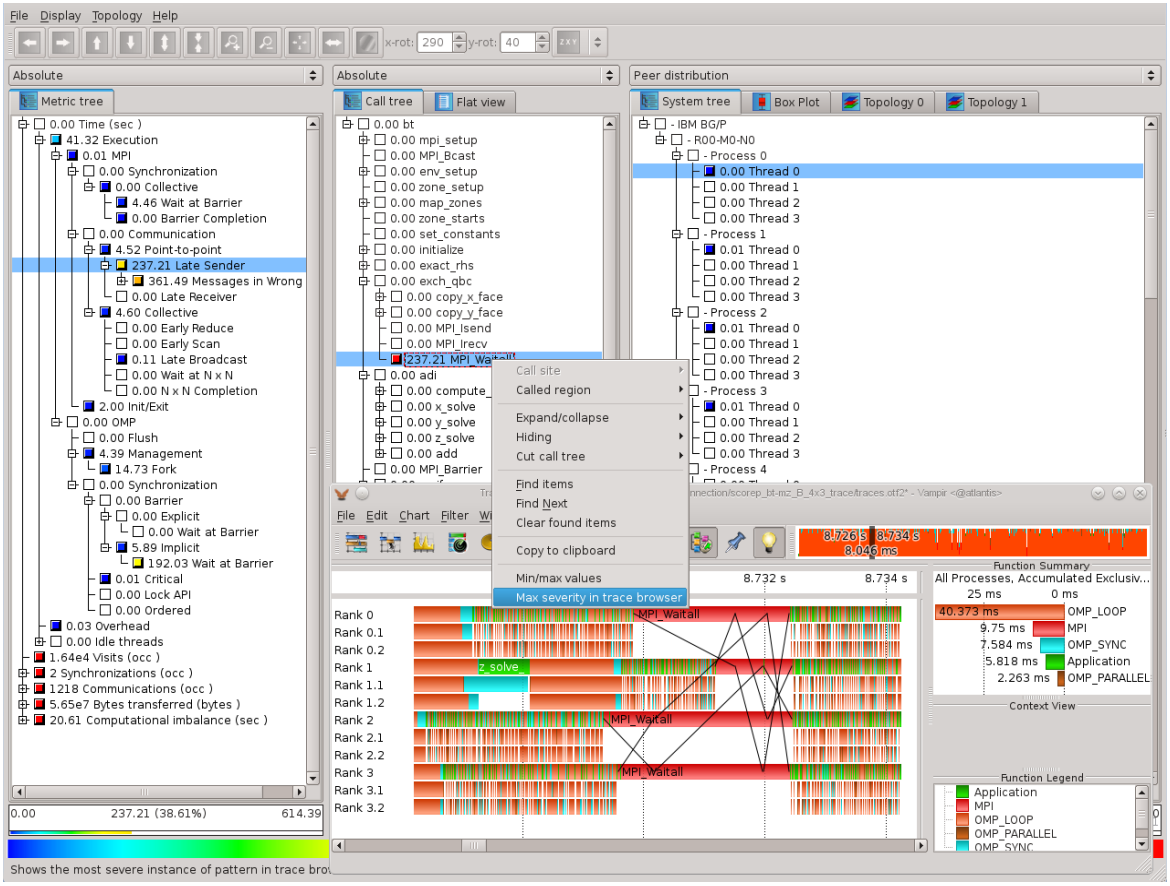

Figure 1.25: Context menu called on a special call path showing the Max severity in trace browser" menu item which results in the location of the worst Late Broadcast instance shown in the timeline display of Vampir. It can be seen that processes enter the `MPI_Bcast` operation earlier than the root process leading to a wait state.

1.6.7 Keyboard and mouse control

1.6.7.1 General control

Shift+F1	Help: What's this?
Ctrl+O	Shortcut for menu `File` ⇒ `Open`
Ctrl+W	Shortcut for menu `File` ⇒ `Close`
Ctrl+Q	Shortcut for menu `File` ⇒ `Quit`
<left-mouse click>	*over menu/tool bar:* activate menu/function *over value mode combo:* select value mode *over tab:* switch to tab *in tree:* select/deselect/expand/collapse items *in topology:* select item
<right-mouse click>	*in tree:* context menu *in topology:* context information
Ctrl+<left-mouse click>	*in tree:* multiple selection/deselection
<left-mouse drag>	*over scroll bar:* scroll *in topology:* rotate topology
Ctrl+<left-mouse drag>	*in topology:* increase plane distance
Shift+<left-mouse drag>	*in topology:* move topology
<scroll mouse-wheel>	*in topology:* zoom in/out
Up arrow	*in tree:* move selection one item up (single-selection only) *in topology/scroll area:* scroll one unit up
Down arrow	*in tree:* move selection one item down (single-selection only) *in topology/scroll area:* scroll one unit down
Left arrow	*in scroll area:* scroll to the left
Right arrow	*in scroll area:* scroll to the right
Page up	*in tree/topology/scroll area:* scroll one page up
Page down	*in tree/topology/scroll area:* scroll one page down

1.6.7.2 Source code editor

Control in read only mode:

Up Arrow	Move one line up
Down Arrow	Move one line down
Left Arrow	Scroll one character to the left (if horizontally scrollable)
Right Arrow	Scroll one character to the right (if horizontally scrollable)
Page Up	Move one (viewport) page up
PageDown	Move one (viewport) page down
Home	Move to the beginning of the text
End	Move to the end of the text
<scroll mouse-wheel>	Scroll the page vertically
Alt+<scroll mouse-wheel>	Scroll the page horizontally (if horizontally scrollable)
Ctrl+<scroll mouse-wheel>	Zoom the text
Ctrl+A	Select all text

Additionally for the read and write mode:

Left Arrow	Move one character to the left
Right Arrow	Move one character to the right
Backspace	Delete the character to the left of the cursor
Delete	Delete the character to the right of the cursor
Ctrl+C	Copy the selected text to the clipboard
Ctrl+Insert	Copy the selected text to the clipboard
Ctrl+K	Delete to the end of the line
Ctrl+V	Paste the clipboard text into text edit
Shift+Insert	Paste the clipboard text into text edit
Ctrl+X	Delete the selected text and copy it to the clipboard
Shift+Delete	Delete the selected text and copy it to the clipboard
Ctrl+Z	Undo the last operation
Ctrl+Y	Redo the last operation
Ctrl+Left arrow	Move the cursor one word to the left
Ctrl+Right arrow	Move the cursor one word to the right
Ctrl+Home	Move the cursor to the beginning of the text
Ctrl+End	Move the cursor to the end of the text
Hold Shift + some movement (e.g., Right arrow)	Select region

1.7 Performance Algebra and Tools

As performance tuning of parallel applications usually involves multiple experiments to compare the effects of certain optimization strategies, CUBE offers a mechanism called *performance algebra* that can be used to merge, subtract, and average the data from different experiments and view the results in the form of a single "derived" experiment. Using the same representation for derived experiments and original experiments provides access to the derived behavior based on familiar metaphors and tools in addition to an arbitrary and easy composition of operations. The algebra is an ideal tool to verify and locate performance improvements and degradations likewise. The algebra includes three operators---*diff*, *merge*, and *mean*---provided as command-line utilities which take two or more CUBE files as input and generate another CUBE file as output. The operations are closed in the sense that the operators can be applied to the results of previous operations. Note that although all operators are defined for any valid CUBE data sets, not all possible operations make actually sense. For example, whereas it can be very helpful to compare two versions of the same code, computing the difference between entirely different programs is unlikely to yield any useful results.

1.7.1 Difference

Changing a program can alter its performance behavior. Altering the performance behavior means that different results are achieved for different metrics. Some might increase while others might decrease. Some might rise in certain parts of the program only, while they drop off in other parts. Finding the reason for a gain or loss in overall performance often requires considering the performance change as a multidimensional structure. With CUBE's difference operator, a user can view this structure by computing the difference between two experiments and rendering the derived result experiment like an original one. The difference operator takes two experiments and computes a derived experiment whose severity function reflects the difference between the minuend's severity and the subtrahend's severity.

The possible output is presented below.

```
user@host: cube_diff scout.cube remapped.cube -o result.cube
Reading scout.cube ... done.
Reading remapped.cube ... done.
++++++++++++ Diff operation begins ++++++++++++++++++++++++++
INFO::Merging metric dimension... done.
INFO::Merging program dimension... done.
INFO::Merging system dimension... done.
INFO::Mapping severities... done.
INFO::Adding topologies...
        Topology retained in experiment.
 done.
INFO::Diff operation... done.
++++++++++++ Diff operation ends successfully +++++++++++++++++
```

```
Writing result.cube ... done.
```

Usage: cube_diff [-o output] [-c] [-C] [-h] minuend subtrahend

-o Name of the output file (default: diff.cube)

-c Do not collapse system dimension, if experiments are incompatible

-C Collapse system dimension

-h Help; Output a brief help message.

1.7.2 Merge

The merge operator's purpose is the integration of performance data from different sources. Often a certain combination of performance metrics cannot be measured during a single run. For example, certain combinations of hardware events cannot be counted simultaneously due to hardware resource limits. Or the combination of performance metrics requires using different monitoring tools that cannot be deployed during the same run. The merge operator takes an arbitrary number of CUBE experiments with a different or overlapping set of metrics and yields a derived CUBE experiment with a joint set of metrics.

The possible output is presented below.

```
user@host: cube_merge scout.cube remapped.cube -o result.cube
++++++++++++ Merge operation begins ++++++++++++++++++++++++++
Reading scout.cube ... done.
Reading remapped.cube ... done.
INFO::Merging metric dimension... done.
INFO::Merging program dimension... done.
INFO::Merging system dimension... done.
INFO::Mapping severities... done.
INFO::Merge operation...
        Topology retained in experiment.

        Topology retained in experiment.
 done.
++++++++++++ Merge operation ends successfully +++++++++++++++++
Writing result.cube ... done.
```

Usage: cube_merge [-o output] [-c] [-C] [-h] cube ...

-o Name of the output file (default: merge.cube)

-c Do not collapse system dimension, if experiments are incompatible

-C Collapse system dimension

-h Help; Output a brief help message.

1.7.3 Mean

The mean operator is intended to smooth the effects of random errors introduced by unrelated system activity during an experiment or to summarize across a range of execution parameters. You can conduct several experiments and create a single average experiment from the whole series. The mean operator takes an arbitrary number of arguments.

The possible output is presented below.

```
user@host: cube_mean scout1.cube scout2.cube scout3.cube scout4.cube -o mean.cube
++++++++++++ Mean operation begins ++++++++++++++++++++++++++
Reading scout1.cube ... done.
INFO::Merging metric dimension... done.
INFO::Merging program dimension... done.
INFO::Merging system dimension... done.
INFO::Mapping severities... done.
INFO::Adding topologies... done.
INFO::Mean operation... done.
Reading scout2.cube ... done.
INFO::Merging metric dimension... done.
INFO::Merging program dimension... done.
INFO::Merging system dimension... done.
INFO::Mapping severities... done.
INFO::Adding topologies... done.
INFO::Mean operation... done.
Reading scout3.cube ... done.
INFO::Merging metric dimension... done.
INFO::Merging program dimension... done.
INFO::Merging system dimension... done.
INFO::Mapping severities... done.
INFO::Adding topologies... done.
INFO::Mean operation... done.
Reading scout4.cube ... done.
INFO::Merging metric dimension... done.
INFO::Merging program dimension... done.
INFO::Merging system dimension... done.
INFO::Mapping severities... done.
INFO::Adding topologies... done.
INFO::Mean operation... done.
++++++++++++ Mean operation ends successfully ++++++++++++++++
Writing mean.cube ... done.
```

Usage: cube_mean [-o output] [-c] [-C] [-h cube ...

-o Name of the output file (default: mean.cube)

-c Do not collapse system dimension, if experiments are incompatible

-C Collapse system dimension

-h Help; Output a brief help message.

1.7.4 Compare

Compares two experiments and prints out if they are equal or not. Two experiments are equal if they have same dimensions hierarchy and the equal values of the severieties.

An example of the output is below.

```
user@host: cube_cmp remapped.cube scout1.cube
Reading remapped.cube ... done.
Reading scout1.cube ... done.
++++++++++++ Compare operation begins ++++++++++++++++++++++++++
Experiments are not equal.
++++++++++++++ Compare operation ends successfully ++++++++++++++++
```

Usage: cube_cmp [-h] cube1 cube2

-h Help; Output a brief help message.

1.7.5 Clean

CUBE files may contain more data in the definition part than absolutely necessary. The `cube_clean` utility creates a new CUBE file with an identical structure as the input experiment, but with the definition part cleaned up.

An example of the output is presented below.

```
user@host: cube_clean remapped.cube -o cleaned.cube
++++++++++++ Clean operation begins ++++++++++++++++++++++++++
Reading remapped.cube ... done.

        Topology retained in experiment.
++++++++++++ Clean operation ends successfully ++++++++++++++++
Writing cleaned.cube ... done.
```

Usage: cube_clean [-o output] [-h cube

-o Name of the output file (default: clean.cube|.gz)

-h Help; Output a brief help message.

1.7.6 Reroot, Prune

For the detailed study of some part of the execution, the CUBE file can be modified based on a given call-tree node. Two different operations are possible:

- The call tree may be re-rooted, i.e., only sub-trees with the given call-tree node as root are retained in the experiment.
- An entire sub-tree may be pruned, i.e., removed from the experiment. In this case, all metric values for that sub-tree will be attributed to it's parent call-tree node (= "inlined").

An example of the output is presented below.

```
user@host: cube_cut -r inner_auto_ -p flux_err_ -o cutted.cube remapped.cube
Reading remapped.cube ... done.
++++++++++++ Cut operation begins ++++++++++++++++++++++++++

        Topology retained in experiment.
++++++++++++ Cut operation ends successfully +++++++++++++++++
Writing cutted.cube ... done.
```

Usage: cube_cut [-h] [-r nodename] [-p nodename] [-o output cube

-o Name of the output file (default: cut.cube|.gz)

-r Re-root call tree at named node

-p Prune call tree from named node (= ”inline”)

-h Help; Output a brief help message.

1.7.7 Remap (version 2)

A more flixible implementation of the tool `cube_remap` is the `cube_remap2`.

This tool takes a remapping specification file as a command line argument and perform recalculation of the metric values according to the specified rules, expressed in CubePL syntax.

This tool can be used to convert all derived metrics into usual metrics, which are holding data (notice, that POSTDERIVED metrics became invalid while this conversion).

CUBE provides examplees of remapping specification files for SCOUT and Score-P. They are stored in the directory `[prefix]/share/doc/cube/examples`

Usage: ./cube_remap2 -r <remap specification file> [-o output] [-d] [-s] [-h] <cube experiment>

-r Name of the remapping specification file. By omitting this option the specification file from the cube experiment is taken if present.

-c Create output file with the same structure as an input file. It overrides option "-r"

-o Name of the output file (default: remap)

-d Convert all prederived metrics into usual metrics, calculate and store their values as a data.

-s Add hardcoded SCALSCA metrics "Idle threads" and "Limited parallelizm"

-h Help; Output a brief help message.

1.7.8 Statistics

Extracts statistical information from the CUBE files.

```
user@host: ./cube_stat  -m time,mpi -p remapped.cube -%
MetricRoutine     Count    Sum       Mean      Variance  Minimum      ...  Maximum
time INCL(MAIN__)     4  143.199101  35.799775  0.001783   35.759769   ...  35.839160
time   EXCL(MAIN__)   4    0.078037   0.019509  0.000441    0.001156   ...   0.037711
time   task_init_     4    0.568882   0.142221  0.001802    0.102174   ...   0.181852
time   read_input_    4    0.101781   0.025445  0.000622    0.000703   ...   0.051980
time   decomp_        4    0.000005   0.000001  0.000000    0.000001   ...   0.000002
time   inner_auto_    4  142.361593  35.590398  0.000609   35.566589   ...  35.612125
time   task_end_      4    0.088803   0.022201  0.000473    0.000468   ...   0.043699

mpi  INCL(MAIN__)     4   62.530811  15.632703  2.190396   13.607989   ...  17.162466
mpi    EXCL(MAIN__)   4    0.000000   0.000000  0.000000    0.000000   ...   0.000000
mpi    task_init_     4    0.304931   0.076233  0.001438    0.040472   ...   0.113223
mpi    read_input_    4    0.101017   0.025254  0.000633    0.000034   ...   0.051952
mpi    decomp_        4    0.000000   0.000000  0.000000    0.000000   ...   0.000000
pi    inner_auto_     4   62.037503  15.509376  2.194255   13.478049   ...  17.031288
mpi    task_end_      4    0.087360   0.021840  0.000473    0.000108   ...   0.043333
```

```
user@host: ./cube_stat  -t33 remapped.cube -p -m time,mpi,visits
Region        NumberOfCalls ExclusiveTime InclusiveTime      time       mpi      visits
sweep_                   48     76.438435    130.972847   76.438435  0.000000      48
MPI_Recv              39936     36.632249     36.632249   36.632249 36.632249   39936
MPI_Send              39936     17.684986     17.684986   17.684986 17.684986   39936
MPI_Allreduce           128      7.383530      7.383530    7.383530  7.383530     128
source_                  48      3.059890      3.059890    3.059890  0.000000      48
MPI_Barrier              12      0.382902      0.382902    0.382902  0.382902      12
flux_err_                48      0.380047      1.754759    0.380047  0.000000      48
TRACING                   8      0.251017      0.251017    0.251017  0.000000       8
MPI_Bcast                16      0.189381      0.189381    0.189381  0.189381      16
MPI_Init                  4      0.170402      0.419989    0.170402  0.170402       4
snd_real_             39936      0.139266     17.824251    0.139266  0.000000   39936
MPI_Finalize              4      0.087360      0.088790    0.087360  0.087360       4
initialize_               4      0.084858      0.168192    0.084858  0.000000       4
initxs_                   4      0.083242      0.083242    0.083242  0.000000       4
MAIN__                    4      0.078037    143.199101    0.078037  0.000000       4
rcv_real_             39936      0.077341     36.709590    0.077341  0.000000   39936
inner_                    4      0.034985    142.337220    0.034985  0.000000       4
inner_auto_               4      0.024373    142.361593    0.024373  0.000000       4
task_init_                4      0.014327      0.568882    0.014327  0.000000       4
read_input_               4      0.000716      0.101781    0.000716  0.000000       4
octant_                 416      0.000581      0.000581    0.000581  0.000000     416
global_real_max_         48      0.000441      1.374712    0.000441  0.000000      48
global_int_sum_          48      0.000298      5.978850    0.000298  0.000000      48
global_real_sum_         32      0.000108      0.030815    0.000108  0.000000      32
barrier_sync_            12      0.000105      0.383007    0.000105  0.000000      12
bcast_int_               12      0.000068      0.189395    0.000068  0.000000      12
timers                    2      0.000044      0.000044    0.000044  0.000000       2
initgeom_                 4      0.000042      0.000042    0.000042  0.000000       4
```

```
initsnc_                4       0.000038      0.000050    0.000038    0.000000       4
task_end_               4       0.000013      0.088803    0.000013    0.000000       4
bcast_real_             4       0.000010      0.000065    0.000010    0.000000       4
decomp_                 4       0.000005      0.000005    0.000005    0.000000       4
timers_                 2       0.000004      0.000048    0.000004    0.000000       2
```

Usage: cube_stat [-h] [-p] [-m metric[,metric...] -%] [-r routine[,routine...] cubefile

OR

cube_stat -h] [-p] [-m metric[,metric...] -t topN cubefile

-h Display this help message

-p Pretty-print statistics (instead of CSV output)

-% Provide statistics about process/thread metric values

-m List of metrics (default: time)

-r List of routines (default: main)

-t Number for topN regions flat profile

1.7.9 from TAU to CUBE

Converts a profile generated by the TAU Performance System [11] into the CUBE format. Currently, only 1-level, 2-level and full call-path profiles are supported.

An example of the output is presented below.

```
user@host: ./tau2cube3 tau2 -o b.cube
    Parsing TAU profile...
    tau2/profile.0.0.2
    tau2/profile.1.0.0
    Parsing TAU profile...          done.
    Creating CUBE profile...
    Number of call paths : 5
    Childmain int (int, char **)
    Number of call paths : 5
    ChildsomeA void (void)
    Number of call paths : 5
    ChildsomeB void (void)
    Number of call paths : 5
    ChildsomeC void (void)
    Number of call paths : 5
    ChildsomeD void (void)
    Path to Parents : 5
    Path to Child : 1
    Number of roots : 5
    Call-tree node created
    Call-tree node created
    Call-tree node created
```

```
Call-tree node created
Call-tree node created
value time :: 8.0151
value ncalls :: 1
value time :: 11.0138
value ncalls :: 1
value time :: 8.01506
value ncalls :: 1
value time :: 11.0138
value ncalls :: 1
value time :: 5.00815
value ncalls :: 1
value time :: 11.0138
value ncalls :: 1
value time :: 0.000287
value ncalls :: 1
value time :: 11.0138
value ncalls :: 1
value time :: 0
value ncalls :: 0
value time :: 9.00879
value ncalls :: 1
done.
```

Usage: tau2cube [tau-profile-dir][-o cube]

1.7.10 Common Calltree

Common Calltree is a tool to reduce call trees of a set of cube files to the common part.

Usage: cube_commoncalltree cubefile1 cubefile2 ... cubefileN

The tool *cube_commoncalltree* takes set of input cubefiles

(*cubefile1 cubefile2 ... cubefileN*)

and creates corresponding set of cube files

(*cubefile1_commoncalltree cubefile2_commoncalltree ... cubefileN_commoncalltree*).

Output cube files *cubefileX_commoncalltree* do have the equal system and metric dimensions like corresponding *cubefileX* file.

Call trees among *cubefileX_commoncalltree* files are reduced to the maximal (up to a special case in region naming scheme) common part. Inclusive value of the "missing" part is added as a exclusive value to its parent (which is a part of common part of call tree)

This tool is particulary useful for comparison of exprerimens with the different recursion depth or with the additional sub call trees depending on some loop iteration index.

1.7.11 Topology Assistant

Topology assistant is a tool to handle topologies in cube files. It is able to add or edit a topology.

Usage: cube_topoassist {**OPTION**} cubefile

The current available options are:

- To create a new topology in an existing cube file,
- To [re]name an existing virtual topology, and
- To [re]name the dimensions of a virtual topology.

The command-line switches for this utility are:

-c: creates a new topology in a given cube file.

-n: displays a numbered list of the existing topologies in the given cube file, and lets the user choose one to be named or renamed.

-d: displays the existing topologies, and lets the user name the dimensions of one of them.

The resulting CUBE file is named topo.cube[.gz], in the current directory.

As mentioned abot, when using the **-d** or **-n** command-line options, a numbered list of the current topologies will appear, showing the topology names, its dimension names (when existing), and the number of coordinates in each dimension, as well as the total number of threads. This is an example of the usage:

```
$ cube_topoassist topo.cube.gz -n
Reading topo.cube.gz . Please wait... Done.
Processes are ordered by rank. For more information about this file,
use cube_info -S <cube experiment>

This CUBE has 3 topologie(s).
0. <Unnamed topology>, 3 dimensions: x: 3, y: 1, z: 4. Total = 12 threads.
1. Test topology, 1 dimensions: dim_x: 12. Total = 12 threads.
2. <Unnamed topology>, 3 dimensions: 3, 1, 4. Total = 12 threads. <Dimensions are not named>


Topology to [re]name?
1
New name:
Hardware topology
Topology successfully [re]named.

Writing topo.cube.gz ... done.
```

The process is similar for [re]naming dimensions within a topology. One characteristic is that either all dimensions are named, or none.

One could easily create a script to generate the coordinates according to some algorithm/equation, and feed this to the assistant as an input. The only requirement is to answer the questions in the order they appear, and after that, feed the coordinates. Coordinates are asked for in rank order, and inside every rank, in thread order.

The sequence of questions made by the assistant when creating a new topology (the **-c** switch) is:

- New topology's name
- Number of dimensions
- Will the above dimensions be named? (Y/N)
- If yes, asks the name. Empty is not valid.
- Number of coordinates in that dimension
- Asks if this dimension is either periodic or not (Y/N)
- Repeat the previous three steps for every dimension
- After that, it expects the coordinates for each thread in this topology, separated by spaces, in the order described above.

This is a sample session of the assistant:

```
$ cube_topoassist -c experiment.cube.gz
Reading experiment.cube.gz. Please wait... Done.
Processes are ordered by rank. For more information about this file, use cube_info -S <cube experiment

So far, only cartesian topologies are accepted.
Name for new topology?
Test topology
Number of Dimensions?
3
Do you want to name the dimensions (axis) of this topology? (Y/N)
y
Name for dimension 0
torque
Number of elements for dimension 0
2000
Is dimension 0 periodic?
y
Name for dimension 1
rotation
Number of elements for dimension 1
1500
Is dimension 1 periodic?
n
Name for dimension 2
period
Number of elements for dimension 2
50
Is dimension 2 periodic?
```

```
n
Alert: The number of possible coordinates (150000000) is bigger than the number of threads
 on the specified cube file (12). Some positions will stay empty.
Topology on THREAD level.
Thread 0's (rank 0) coordinates in 3 dimensions, separated by spaces
0 0 0
0 0 1
0 0 2
...
...
...
Writing topo.cube.gz ... done.
$
```

So, a possible input file for this cube experiment could be:

```
Test topology
3
y
torque
2000
y
rotation
1500
n
period
50
n
0 0 0
0 0 1
0 0 2
... (the remaining coordinates)
```

And then call the assistant: cube_topoassist -c cubefile.cube < input.txt

1.7.12 Dump

To export values from the cube report into another tool or to examine internal structure of the cube report CUBE framework provedes a tool `cube_dump` tool, which prints out different values. It calculates inclusive and exclusive values along metric tree and call tree, agregates over system tree or displays values for every thread separately. In addition it provides user to define new metrics(see file CubeDerivedMetrics.pdf). Results are calculated and shown. For convenience user can invoke defined metrics along with new once in any order. For doing so one lists unique names of metrics separated by commas. For access to more than one callpaths, user can specify the ids or a range of them like "2-9". This also can be done for threads. Additionally provides a calculation of the flat profile values.

-m <**metrics**>|<**new metrics**>|**all**|<**filename**> Select one or more of metrics (unique names) for data dump.

By giving a CubePL expression one can define one or more new metrics by giving correspond formula. If the expression prefixed with "<name>:", <name> is used as a unique name for the new metric.

<filename> - takes a CubePL expression from file <filename> and defines a derived metric with it

all - all metrics will be printed.

Combination of these three is possible.

-c <**cnode ids**>|**all**|**leafs**|**roots**|**level**>**X**|**level=X**|**level**<**X**|**name=/regexp/**|<**filename**>

Select one or more call paths to be printed out .

<cnode ids> - list or range of call path ids: 0,3,5-10,25

all - all call paths are printed

leafs - only call paths without children are printed

roots - only root cnodes are printed

level<X, level=X or level>X - only cnodes of the level more, equal or less than N are printed

name=/regexp/- only cnodes with the region name matching to the regular expression *regexp*

<filename> - takes a CubePL expression from file <filename> and evaluates it for every callpath.

If the result is non-zero - call path is included into the output.

-x incl|**excl** Selects, if the data along the metric tree should be calculated as an inclusive or an exclusive value.

(Default value: incl).

-z incl|**excl**|**stored** Selects, if the data along the call tree should be calculated as an inclusive or an exclusive value.

(Default value: excl).

-t <**thread id**>|**aggr** Show data for one or more selected threads or aggregated over system tree.

-r Prints aggregated values for every region (flat profile), sorted by id.

-f <**name**> Selects a stored data with the name <name> to display

-d Shows the coordinates for every topology as well.

-y Disables expansion of clusters and shows bare stored meta structure.

-w Prints out the information about structure of the cube.

-o <**filename**>|**-** Uses a device or STDOUT for the output. If omit, STDOUT is used.

-s human|**gnuplot**|**csv**|**csv2**|**R** Uses either human readable form, GNUPLOT or CSV (two different layouts) format or binary R matrix for data export.

-h|**-?** Help; Output a brief help message.

This is examples of the usage.

Example 1:

```
$cube_dump -m time,"metric::visits(e)","metric::time(i)/metric::visits(e)" -c 0 \
  -z incl -s gnuplot  profile.cubex

# ====================== DATA ==================
# Print out the data of the metric time
#main(id=0)
0   0   80.549003343
0   1   44.115097986
0   2   43.486614165
0   3   43.940738098
0   4   80.539393011
0   5   42.723353088
0   6   42.61159706
0   7   43.108220977
0   8   80.635220741
0   9   43.788284208
0   10  43.831524441
0   11  43.652044759
0   12  80.629776666
0   13  42.692885677
0   14  42.719330066
0   15  42.732487708

# Print out the data of the metric New Metric1
#main(id=0)
0   0   80.549003343
0   1   1.79769313486e+308
0   2   1.79769313486e+308
0   3   1.79769313486e+308
0   4   80.539393011
0   5   1.79769313486e+308
0   6   1.79769313486e+308
0   7   1.79769313486e+308
0   8   80.635220741
0   9   1.79769313486e+308
0   10  1.79769313486e+308
0   11  1.79769313486e+308
0   12  80.629776666
0   13  1.79769313486e+308
0   14  1.79769313486e+308
```

```
0   15  1.79769313486e+308

# Print out the data of the metric New Metric2
#main(id=0)
0   0   1
0   1   0
0   2   0
0   3   0
0   4   1
0   5   0
0   6   0
0   7   0
0   8   1
0   9   0
0   10  0
0   11  0
0   12  1
0   13  0
0   14  0
0   15  0
```

Example 2:

```
$cube_dump -m time,"metric::visits(e)","metric::time(i)/metric::visits(e)" -c 0 \
 -t aggr  -z incl -s human  profile.cubex

 ====================== DATA ===================
Print out the data of the metric time

            All threads
-------------------------------------------------------------------------------
main(id=0)  841.755571994
Print out the data of the metric New Metric1

            All threads
-------------------------------------------------------------------------------
main(id=0)  210.438892999
Print out the data of the metric New Metric2

            All threads
-------------------------------------------------------------------------------
main(id=0)  4
```

Example 3:

```
$cube_dump -m time,"metric::visits(e)","metric::time(i)/metric::visits(e)" -c 20 \
  -z incl -s csv  profile.cubex

20,0,8.9812e-05
20,1,0
```

```
20,2,0
20,3,0
20,4,9.7463e-05
20,5,0
20,6,0
20,7,0
20,8,0.000132327
20,9,0
20,10,0
20,11,0
20,12,7.2788e-05
20,13,0
20,14,0
20,15,0

20,0,8.9812e-05
20,1,-1.79769313486e+308
20,2,-1.79769313486e+308
20,3,-1.79769313486e+308
20,4,9.7463e-05
20,5,-1.79769313486e+308
20,6,-1.79769313486e+308
20,7,-1.79769313486e+308
20,8,0.000132327
20,9,-1.79769313486e+308
20,10,-1.79769313486e+308
20,11,-1.79769313486e+308
20,12,7.2788e-05
20,13,-1.79769313486e+308
20,14,-1.79769313486e+308
20,15,-1.79769313486e+308

20,0,1
20,1,0
20,2,0
20,3,0
20,4,1
20,5,0
20,6,0
20,7,0
20,8,1
20,9,0
20,10,0
20,11,0
20,12,1
20,13,0
20,14,0
20,15,0
```

Example 4:

```
$cube_dump -m time,"metric::visits(e)","metric::time(i)/metric::visits(e)" -c 1 \
```

```
  -z incl -s csv2  profile.cubex

Cnode ID, Thread ID,time,New Metric1,New Metric2
1,0,80.548967177,80.548967177,1
1,1,44.115097986,1.79769313486e+308,0
1,2,43.486614165,1.79769313486e+308,0
1,3,43.940738098,1.79769313486e+308,0
1,4,80.539359524,80.539359524,1
1,5,42.723353088,1.79769313486e+308,0
1,6,42.61159706,1.79769313486e+308,0
1,7,43.108220977,1.79769313486e+308,0
1,8,80.635176341,80.635176341,1
1,9,43.788284208,1.79769313486e+308,0
1,10,43.831524441,1.79769313486e+308,0
1,11,43.652044759,1.79769313486e+308,0
1,12,80.629742485,80.629742485,1
1,13,42.692885677,1.79769313486e+308,0
1,14,42.719330066,1.79769313486e+308,0
1,15,42.732487708,1.79769313486e+308,0
```

Example 5:

```
$cube_dump -m time,"metric::visits(e)","metric::time(i)/metric::visits(e)" -c 1 \
  -z incl -s R  profile.cubex -o output_file
```

This will generate binary file "output_file" which can be loaded in R. In consists of three matrices, each one corresponding to one metric. Each matrix is named after the metric and it contains values for all threads and nodes.

Example 6: We select only call path names, starting with "main" using the CubePL expression (stored in file name "selection.cubepl") :

```
{
${a}=0;
if (${cube::region::name}[${calculation::region::id}] =~  /^main/ )
{ ${a}=1; }; return ${a};
}
```

Then:

```
cube_dump -m time,"metric::visits(e)","metric::time(i)/metric::visits(e)"   -c selection.cubepl  \
 -z incl  -t aggr  profile.cubex

 ====================== DATA ==================
Print out the data of the metric time
                  All threads
-------------------------------------------------------------------------------
main(id=0)        841.755571994
main_loop(id=12)  840.73706946

Print out the data of the metric New Metric1
```

```
                  All threads
-------------------------------------------------------------------------------
main(id=0)        210.438892999
main_loop(id=12)  0.210184267365

Print out the data of the metric New Metric2
                  All threads
-------------------------------------------------------------------------------
main(id=0)        4
main_loop(id=12)  4000
```

Options "leafs", "roots", "level=", "level<", "level>" and "name=/regexp/" are shortcuts for a build-in CubePL expression, which is used to select a call path.

- **"leafs"** : stands for:

```
{
${a}=0;
if (${cube::callpath::#children}[${calculation::callpath::id}] == 0 )
{ ${a}=1; }; return ${a};
}
```

- **"roots"** : stands for:

```
{
${a}=0;
if (${cube::callpath::parent::id}[${calculation::callpath::id}] == -1 )
{ ${a}=1; }; return ${a};
}
```

- **"level=N"** : stands for:

```
{
${level}=N;
${index}=0;
${i}=${calculation::callpath::id};
while (${cube::callpath::parent::id}[${i}] != -1 )
{ ${i}= ${cube::callpath::parent::id}[${i}]; ${index}=${index}+1; };
${a}=0;
if (${index} ==  ${level})
{${a}=1; };
return ${a};
}
```

- **"name=/regexp/"** : stands for:

```
{
  ${a}=0;
   if ( ${cube::region::name}[${calculation::region::id}] =~  /regexp/ )
  { ${a}=1; }; return ${a};
};
```

"level<N" and "level>N" differ from "level=N" in the boolean operation in the line 8. For detailed documentation of the syntax of CubePL please see[12].

2 CUBE4 API

2.1 Creating CUBE Files

The CUBE data format in a tar envelope, having extension `.cubex`, with various files. Description of the cube stored in file `anchor.xml`[10] inside of the `cubex` instance. The data stored in binary format in various files inside of the `cubex` file. The CUBE library provides an interface to create CUBE files. It is a simple class interface and includes only a few methods. This section first describes the CUBE APIand then presents a simple C++ program as an example of how to use it.

2.1.1 CUBE API

The class interface defines a class `Cube`. The class provides a default constructor and nearly forty methods. The methods are divided into five groups. The first three groups are used to define the three dimensions of the performance space, forth group is used to enter the actual data and last group is used to open or to save the cube report. In addition, an output operator $<<$ to export the data into CUBE3 format is provided.

2.1.1.1 Metric Dimension

This group refers to the metric dimension of the performance space. It consists of a single method used to build metric trees. Each node in the metric tree represents a performance metric. Metrics have different units of measurement. The unit can be either "`sec`" (i.e., seconds) for time based metrics, such as execution time, or "`occ`" (i.e., occurrences) for event-based metrics, such as floating-point operations. During the establishment of a metric tree, a child metric is usually more specific than its parent, and both of them have the same unit of measurement. Thus, a child performance metric has to be a subset of its parent metric (e.g., system time is a subset of execution time).

```
Metric* def_met(const std::string &disp_name, const std::string &uniq_name,
                const std::string &dtype, const std::string &uom,
                const std::string &val, const std::string &url,
                const std::string &descr, Metric* parent,
                TypeOfMetric       type_of_metric = CUBE_METRIC_EXCLUSIVE,
                const std::string& expression = "",
                TypeOfMetric       is_ghost = CUBE_METRIC_NO_GHOST);
```

Returns a metric with display name `disp_name`, unique name `uniq_name` and description `descr`.

`dtype` specifies the data type, which can either be "`INTEGER`" or "`FLOAT`".

`uom` is the unit of measurement, which is either "`sec`" for seconds or "`occ`" for number of occurrences.

`val` specifies whether there is any data available for this particular metric. It can either be "`VOID`" (no data available, metric will not be shown in CUBE) or an empty string (metric will be shown and data is present).

`parent` is a previously created metric which will be the new metric's parent. To define a root node, use `NULL` instead.

`url` is a link to an HTMLpage describing the new metric in detail. If you want to mirror the page at several locations, you can use the macro `@mirror@` as a prefix, which will be replaced by an available mirror defined using `def_mirror()` (see Section**??**).

`type_of_metric` specifies the nature of this metric. If you want to store exclusive (along call tree) values, use a constant CUBE_METRIC_EXCLUSIVE, if you want to store inclusive (along call tree) values, use a constant CUBE_METRIC_EXCLUSIVE, if you want to define a derived metric, use one of the constants: CUBE_POSTDERIVED_METRIC, CUBE_PREDERIVED_METRIC_EXCLUSIVE or CUBE_PREDERIVED_METRIC_INCLUSIVE.

`expression` is a CubePL expression to specify the derived metric.

`is_ghost` is used internally by CubePL Engine and can be left with default value `CUBE_METRIC_NO_GHOST`

For further information about kinds of the derived metrics in cube and about CubePL syntax see[12].

```
const std::vector<Metric*>& get_metv() const;
```

Returns a vector with all metrics in the CUBE object.

```
const std::vector<Metric*>& get_root_metv() const;
```

Returns a vector with all roots of the metric dimension in the CUBE object.

```
Metric* get_met( const std::string& uniq_name) const;
```

Returns a metric with the given `uniq_name`. Returns `NULL` if the CUBE object doesn't contain a metric with this name.

```
 Metric* get_root_met( Metric * met);
```

Returns the root metric for the given metric `met`.

2.1.1.2 Program Dimension

This group refers to the program dimension of the performance space. The entities presented in this dimension are *region*, *call site*, and *call-tree node* (i.e., call paths). A region can be a function, a loop, or a basic block. Each region can have multiple call sites from which the control flow of the program enters a new region. Although we use the term call site here, any place that causes the program to enter a new region can be represented as a call site, including loop entries. Correspondingly, the region entered from a call site is called *callee*, which might as well be a loop. Every call-tree node points to a call site. The actual call path represented by a call-tree node can be derived by following all the call sites starting at the root node and ending at the particular node of interest. The user can choose among three ways of defining the program dimension:

1. Call tree with line numbers
2. Call tree without line numbers
3. Flat profile

A call tree with line numbers is defined as a tree whose nodes point to call sites. A call tree without line numbers is defined as a tree whose nodes point to regions (i.e., the callees). A flat profile is simply defined as a set of regions, that is, no tree has to be defined.

```
Region* def_region
              (const std::string &name, long begln, long endln,
              const std::string &url, const std::string &descr,
              const std::string &mod);
```

Returns a new region with region name `name` and description `descr`. The region is located in the module `mod` and exists from line `begln` to line `endln`.

`url` is a link to an HTMLpage describing the new region in detail. For example, if the region is a library function, the `url` can point its documentation. If you want to mirror the page at several locations, you can use the macro `@mirror@` as a prefix, which will be replaced by an available mirror defined using disp{def_mirror()} (see Section**??**).

```
Cnode* def_cnode
             (Region* callee,
             const std::string &mod, int line,
             Cnode* parent);
```

Returns a new call-tree node representing a call from call site located at the line `line` of the module `mod`. The call tree node calls the callee `callee` (i.e., a previously defined region). `parent` is a previously created call-tree node which will be the new one's parent. To define a root node, use `NULL` instead. This method is used to create a call tree with line numbers.

```
Cnode* def_cnode
           (Region* region,
           Cnode* parent);
```

Defines a new call-tree node representing a call to the region `region`. `parent` is a previously created call-tree node which will be the new one's parent. To define a root node, use `NULL` instead. Note that different from the previous `def_cnode()`, this method is used to create a call-tree without line numbers where each call-tree node points to a region.

To define a call tree with line numbers use `def_cnode(Region*, string, int ...)`. To define a call tree without line numbers use `def_cnode(Region*, Cnode*)` instead. To create a flat profile use neither one --- just defining a set of regions will be sufficient.

```
const std::vector<Region*>& get_regv() const;
```

Returns a vector with all regions in the CUBE object.

```
const std::vector<Cnode*>& get_cnodev() const;
```

Returns a vector with all call-tree nodes in the CUBE object.

```
Cnode* get_cnode(Cnode & cn) const;
```

Search a call-tree node `cn`. Returns `NULL` if the CUBE object does not contain the given call-tree node.

2.1.1.3 System Dimension

This group refers to the system dimension of the performance space. It reflects the system resources which the program is using at runtime. The entities present in this dimension are *machine*, *node*, *process*, *thread*, which populate four levels of the system hierarchy in the given order. That is, the first level consists of machines, the second level of nodes, and so on. Finally, the last (i.e., leaf) level is populated only by threads. The system tree is built in a top-down way starting with a machine. Note that even if every process has only one thread, users still need to define the thread level.

```
Machine* def_mach  (const std::string &name, const std::string &desc);
```

Returns a new machine with the name `name` and description `desc`.

```
Node* def_node (const std::string &name, Machine* mach);
```

Returns a new (SMP) node which has the name `name` and which belongs to the machine `mach`.

```
Process* def_proc
               (const std::string &name, int rank,
               Node* node);
```

Returns a new process which has the name `name` and the rank `rank`. The rank is a number from 0 to $(n-1)$, where n is the total number of processes. MPIapplications may use the rank in `MPI_COMM_WORLD`. The process runs on the node `node`.

```
Thread* def_thrd
              (cosnt std::string name&, int rank,
             ~Process* proc);
```

Defines a new thread which has the name `name` and the rank `rank`. The rank is a number from 0 to $(n-1)$, where n is the total number of threads spawned by a process. Open MPapplications may use the Open MPthread number. The thread belongs to the process `proc`.

```
const std::vector<Sysres*>& get_sysv() const;
```

Returns a vector with all system resources (e.g. node, thread, process) available in the CUBE object.

```
const std::vector<Machine*>& get_machv() const;
```

Returns a vector with all machines in the CUBE object.

```
const std::vector<Node*>& get_nodev() const;
```

Returns a vector with all nodes of all machines in the CUBE object.

```
const std::vector<Process*>& get_procv() const;
```

Returns a vector with all processes in the CUBE object.

```
const std::vector<Thread*>& get_thrdv() const;
```

Returns a vector with all threads in the CUBE object.

```
Machine * get_mach(Machine & mach) const;
```

Search for the machine `mach` in the CUBE object. Returns `NULL` if the CUBE object does not contain the given machine.

```
Node *get_node(Node & node) const;
```

Search for the node `node` in the CUBE object. Returns `NULL` if the CUBE object does not contain the given node.

2.1.1.4 Virtual Topologies

Virtual topologies are used to describe adjacency relationships among machines, SMPnodes, processes or threads. A topology usually consists of a single class of entities such as threads or processes. The CUBE APIprovides a set of functions to create Cartesian topologies and to define the machine/ SMPnode/process/thread mappings onto coordinates. Note that the definition of virtual topologies is optional.

```
Cartesian* def_cart
           (long ndims, const std::vector<long>& dimv,
            const std::vector<bool>& periodv);
```

Defines a new Cartesian topology. `ndims` and `dimv` specify the number of dimensions and the size of each dimension. `periodv` specifies the periodicity for each dimension. Currently,the maximum value for `ndims` is three.

```
void def_coords
           (Cartesian* cart, Sysres* sys,
            const std::vector<long>& coordv);
```

Maps a specific system resource onto a Cartesian coordinate. The system resource `sys` may be a machine, SMPnode, process or a thread. It is not recommended to map a mixed set of entities onto one topology (e.g., machines and threads are located in the same topology). The parameter of `cart` has been defined by the above `def_cart()` method.

```
const std::vector<Cartesian *>& get_cartv () const;
```

Returns a vector of all cartesian topologies available in the CUBE object.

```
const Cartesian * get_cart ( int i) const;
```

Returns in i-th topology in the CUBE object.

2.1.1.5 Severity Mapping

After the establishment of the performance space, users can assign severity values to points of the space. Each point is identified by a tuple (`met`, `cnode`, `thrd`) . The value should be inclusive with respect to the metric, but exclusive with respect to the call-tree node, that is it should not cover its children. The default severity value for the data points left undefined is zero. Thus, users only need to define non-zero data points.

```
void set_sev
      (Metric* met, Cnode* cnode,
      Thread* thrd, double value);
```

Assigns the value `value` to the point (`met`, `cnode`, `thrd`).

```
void add_sev
     (Metric* met, Cnode* cnode,
     Thread* thrd, double value);
```

Adds the value `value` to the present value at point (`met`, `cnode`, `thrd`).

The previous two methods `set_sev()` and `add_sev()` are intended to be used when the program dimension contains a call tree and not a flat profile. As the flat profile does not require the definition of call-tree nodes, the following two functions should be used instead:

```
void set_sev
     (Metric* met, Region* region,
      Thread* thrd, double value);
```

Assigns the value `value` to the point (`met`, `region`, `thrd`).

```
void add_sev
     (Metric* met, Region* region,
      Thread* thrd, double value);
```

Adds the value `value` to the present value at point (`met`, `region`, `thrd`).

```
double get_sev ( Metric * met, Cnode * cnode, Thread * thrd) const;
```

Returns the value for the point (`met`, `cnode`, `thrd`).

Cube library provides various calls of type `get_sev`, which allow to perform different ways of aggregations.

Here is the short list

```
double get_sev ( Metric * met, CalculationFlavour mf,
                 Cnode * cnode, , CalculationFlavour cf,
                 Thread * thrd, CalculationFlavour sf) const;

double get_sev ( Metric * met, CalculationFlavour mf,
                 Region * region, , CalculationFlavour rf,
                 Thread * thrd, CalculationFlavour sf) const;

double get_sev ( Metric * met, CalculationFlavour mf,
                 Cnode * cnode, , CalculationFlavour cf) const;

double get_sev ( Metric * met, CalculationFlavour mf,
                 Region * region, , CalculationFlavour rf) const;


double get_sev ( Metric * met, CalculationFlavour mf) const;
```

With `CalculationFlavour` one calculates either inclusive or exclusive value along the corresponding tree. Value `cube::CUBE_CALCULATE_EXCLUSIVE` stands for exclusive value and value `cube::CUBE_CALCULATE_INCLUSIVE` - for inclusive.

2.1.1.6 Miscellaneous

Often users may want to define some information related to the CUBE file itself, such as the creation date, experiment platform, and so on. For this purpose, CUBE allows the definition of arbitrary attributes in every CUBE data set. An attribute is simply a key-value pair and can be defined using the following method:

```
void def_attr  (const std::string &key, const std::tring &value);
```

Assigns the value `value` to the attribute `key`.

CUBE allows using multiple mirrors for the online documentation associated with metrics and regions. The `url` expression supplied as an argument for `def_metric()` and `def_region()` can contain a prefix `@mirror@`. When the online documentation is accessed, CUBE can substitute all mirrors defined for the prefix until a valid one has been found. If no valid online mirror can be found, CUBE will substitute the `./doc` directory of the installation path for `@mirror@`.

```
void def_mirror  (const std::string &mirror);
```

Defines the mirror `mirror` as potential substitution for the URLprefix `@mirror@`.

```
std::string get_attr(const std::string &key) const;
```

Returns the attribute in the CUBE object stored for the given `key`.

```
const std::map<std::string, std::string>  get_attrs() const;
```

Returns all attributes associated to the CUBE object as a map.

```
const std::vector<std::string>& get_mirrors() const;
```

Returns all mirrors defined in the CUBE object.

```
int get_num_thrd() const;
```

Returns the maximal number of threads per process in the CUBE object.

```
void setGlobalMemoryStrategy( CubeStrategy strategy);
```

Sets same memory usage `strategy` for all metrics. Possible values are:

- CUBE_MANUAL_STRATEGY
- CUBE_ALL_IN_MEMORY_STRATEGY
- CUBE_LAST_N_ROWS_STRATEGY

```
void setMetricMemoryStrategy( Metric* metric, CubeStrategy strategy);
```

Sets memory usage `strategy` for selected metric.

```
void dropRowInMetric( Metric* metric, Cnode * cnode);
```

Removes data row for the `cnode` from the memory of `metric` if its memory straregy allows.

In case of `CUBE_MANUAL_STRATEGY` it is allways the case.

In case of `CUBE_ALL_IN_MEMORY_STRATEGY` it is never the case and this call has no action.

```
void dropRowInAllMetrics(Cnode * cnode);
```

Removes data row for the `cnode` from the memory of all metrics if their memory straregy allows.

```
void reroot_cnode(Cnode * cnode);
```

Removes all parents of the `cnode` and sets it as a root.

```
void prune_cnode(Cnode * cnode);
```

Removes the cnode and its subtree and sets its parent as a leaf.

```
void set_cnode_as_leaf(Cnode * cnode);
```

Removes its subtree.

```
void set_statistics_name(const std::string& name);
```

Stores the name of the statistic file inside of cube report.

```
std::string  get_statistics_name() const;
```

Returns the name of the statistic file if stored.

```
void enable_flat_tree(const bool status);
```

Enables or disables the calculation of the flat tree. For some applications flat tree doesn't make sense.

```
bool is_flat_tree_enabled() const;
```

Returns whether calculation of flat tree is enabled or disabled.

```
void set_metrics_title(const std::string& title);
```

Sets the title for the metric dimension. In some applications with CUBE name `metric` is misleading.

```
std::string&
set_metrics_title() const;
```

Returns the title of the metric dimension.

```
void set_calltree_title(const std::string& title);
```

Sets the title for the program dimension. In some applications with CUBE name `calltree` is misleading.

```
std::string&
set_calltree_title() const;
```

Returns the title of the program dimension.

```
void set_systemtree_title(const std::string& title);
```

Sets the title for the system dimension. In some applications with CUBE name `System` is misleading.

```
std::string&
set_systemtree_title() const;
```

Returns the title of the system dimension.

```
vector<string>
get_misc_data();
```

Returns a list with names of all files, stored inside of the `cubex` container. This list includes files with description of the cube and metric data.

```
vector<char>
get_misc_data(const std::string& name);
```

Returns content of the file `name` if present, otherwise empty vector.

```
void
write_misc_data(const std::string& name,
                const chat* buffer,
                size_t   length);
```

Writes content of the `buffer` of `length` chars as a file with a name `name`.

```
void
write_misc_data(const std::string& name,
                std:vector<char> buffer);
```

Alternatice call to previous.

2.1.1.7 Writer Library in C

In order to create data files, another possibility is to use the C version of the CUBE writer API. The interface defines a `struct cube_t` and provides the following functions:

```
cube_t* cube_create();
```

Returns a new CUBE structure.

```
void cube_free(cube_t* c);
```

Destroys the given CUBE structure.

```
cube_metric* cube_def_met
     (cube_t* c, const char* disp_name,
     const char* uniq_name, const char* dtype,
     const char* uom, const char* val,
     const char* url, const char* descr,
     cube_metric* parent);
```

Returns a new metric structure.

```
cube_region* cube_def_region
                          (cube_t* c, const char* name, long begln,
                          long endln, ~const char* url,
                          const char* descr, const char* mod);
```

Returns a new region.

```
cube_cnode* cube_def_cnode_cs
                    (cube_t* c, cube_region* callee,
                    const char* mod, int line,
                    cube_cnode* parent);
```

Returns a new call-tree node structure with line numbers.

```
cube_cnode* cube_def_cnode
                       (cube_t* c, cube_region* callee,
                        cube_cnode* parent);
```

Returns a new call-tree node structure without line numbers.

```
cube_machine* cube_def_mach
                         (cube_t* c, const char* name
                          const char* desc);
```

Returns a new machine.

```
cube_node* cube_def_node
                (cube_t* c, const char* name,
                cube_machine* mach);
```

Returns a new node.

```
cube_process* cube_def_proc
                    (cube_t* c, const char* name,
                     int rank, cube_node* node);
```

Returns a new process.

```
cube_thread* cube_def_thrd
                    (cube_t* c, const char* name,
                     int rank, cube_process* proc);
```

Returns a new thread.

```
cube_cartesian* cube_def_cart
                    (cube_t* c, long ndims,
                    long int* dimv, int* periodv);
```

Defines a new Cartesian topology.

```
void cube_def_coords
                (cube_t* c, cube_cartesian* cart,
                 cube_thread* thrd, long int* coord);
```

Maps a thread onto a Cartesian coordinate.

```
void cube_set_sev
               (cube_t* c, cube_metric* met, cube_cnode* cnode,
                cube_thread* thrd, double value);
```

Assigns the severity `value` to the point (`met`, `cnode`, `thrd`). Can only be used after metric, cnode and thread definitions are complete. Note that you can only use either the region or the cnode form of these calls, but not both at the same time.

```
double cube_get_sev
                (cube_t* c, cube_metric* met, cube_cnode* cnode,
                 cube_thread* thrd);
```

Returns the severity of the point (`met`, `cnode`, `thrd`).

```
void cube_set_sev_reg
                  (cube_t* c, cube_metric* met, cube_region* reg,
                   cube_thread* thrd, double value);
```

Assigns the severity `value` to the point (`met`, `reg`, `thrd`). Can only be used after metric, regino and thread definitions are complete. Note that you can only use either the region or the cnode form of these calls, but not both at the same time.

```
void cube_add_sev
               (cube_t* c, cube_metric* met, cube_cnode* cnode,
                cube_thread* thrd, double value);
```

Adds the severity `value` to the present value at point (`met`, `cnode`, `thrd`). Can only be used after metric, cnode and thread definitions are complete. Note that you can only use either the region or the cnode form of these calls, but not both at the same time.

```
void cube_add_sev_reg
                 (cube_t* c, cube_metric* met, cube_region* reg,
                  cube_thread* thrd, double value);
```

Adds the severity `value` to the present value at point (`met`, `reg`, `thrd`). Can only be used after metric, region and thread definitions are complete. Note that you can only use either the region or the cnode form of these calls, but not both at the same time.

```
void cube_write_all
             (cube_t* c, FILE* fp);
```

Writes the entire CUBE data to the given file. This basically corresponds to calling `cube_write_def()` and `cube_write_sev_matrix()`.

```
void cube_write_def
             (cube_t* c, FILE* fp);
```

Writes the definitions part of the CUBE data to the given file. Should only be used after definitions are complete.

```
void cube_write_sev_matrix
                  (cube_t* c, FILE* fp);
```

Writes the severity values part of the CUBE data to the given file. Should only be used after severity values are completely set. Unset values default to zero.

```
void cube_write_sev_row
                 (cube_t* c, FILE* fp,
                  cube_metric* met,
                  cube_cnode* cnode,
                  double* sevs);
```

Writes the given severity values of (`met`, `cnode`) for all threads to the given file. This can be used instead of `cube_write_sev_matrix()` to incrementally write parts of the severity matrix.

```
void cube_write_finish
      (cube_t* c, FILE* fp);
```

Writes the end tags to a file. Must be called at the very end before closing the file, but only when incrementally writing the severity matrix using `cube_write_sev_matrix()`. When using `cube_write_sev_matrix()` to write the severity matrix in one chunk, calling this function is not needed.

2.1.2 Typical Usage

A simple C++ program is given to demonstrate how to use the CUBE write interface. Example below shows the corresponding CUBE display. The source code of the target application is provided below.

```
1         void foo() {
              ...
10        }
11        void bar() {
              ...
20        }
21        int main(int argc, char* argv) {
              ...
60          foo();
              ...
80          bar();
              ...
100       }
```

```
// A C++ example using CUBE write interface
#include <cube3/Cube.h>
#include <string>
#include <fstream>

using namespace std;
using namespace cube;

int main(int argc, char* argv[]) {
  Cube cube;

  // Specify mirrors (optional)
  cube.def_mirror("http://icl.cs.utk.edu/software/kojak/");
  cube.def_mirror("http://www.fz-juelich.de/jsc/kojak/");

  // Specify information related to the file (optional)
  cube.def_attr("experiment time", "September 27th, 2006");
  cube.def_attr("description", "a simple example");

  // Build metric tree
  Metric* met0 = cube.def_met("Time", "Time", "FLOAT", "sec", "",
                              "@mirror@patterns-2.1.html#execution",
                              "root node", NULL); // using mirror
  Metric* met1 = cube.def_met("User time", "User Time", "FLOAT", "sec", "",
                              "http://www.cs.utk.edu/usr.html",
                              "2nd level", met0); // without using mirror
  Metric* met2 = cube.def_met("System time", "System Time", "FLOAT", "sec", "",
                              "http://www.cs.utk.edu/sys.html",
                              "2nd level", met0); // without using mirror

  // Build call tree
```

```
string mod  = "/ICL/CUBE/example.c";
Region* regn0  = cube.def_region("main", 21, 100, "", "1st level", mod);
Region* regn1  = cube.def_region("foo", 1, 10, "", "2nd level", mod);
Region* regn2  = cube.def_region("bar", 11, 20, "", "2nd level", mod);

Cnode* cnode0 = cube.def_cnode(regn0, mod, 21, NULL);
Cnode* cnode1 = cube.def_cnode(regn1, mod, 60, cnode0);
Cnode* cnode2 = cube.def_cnode(regn2, mod, 80, cnode0);

// Build system resource tree
Machine* mach  = cube.def_mach("MSC", "");
Node*    node  = cube.def_node("Athena", mach);
Process* proc0 = cube.def_proc("Process 0", 0, node);
Process* proc1 = cube.def_proc("Process 1", 1, node);
Thread*  thrd0 = cube.def_thrd("Thread 0", 0, proc0);
Thread*  thrd1 = cube.def_thrd("Thread 1", 1, proc1);

// Build 2D Cartesian a topology (a 5x5 grid)
int ndims = 2;
vector<long> dimv;
vector<bool> periodv;
for (int i = 0; i < ndims; i++) {
  dimv.push_back(5);
  if (i % 2 == 0)
    periodv.push_back(true);
  else
    periodv.push_back(false);
}
Cartesian* cart = cube.def_cart(ndims, dimv, periodv);
vector<long> coord0, coord1;
coord0.push_back(0);
coord0.push_back(0);
coord1.push_back(3);
coord1.push_back(3);
// map the two threads onto the above 2 coordinates
cube.def_coords(cart, thrd0, coord0);
cube.def_coords(cart, thrd1, coord1);

// Severity mapping
cube.set_sev(met0, cnode0, thrd0, 4);
cube.set_sev(met0, cnode0, thrd1, 4);
cube.set_sev(met0, cnode1, thrd0, 4);
cube.set_sev(met0, cnode1, thrd1, 4);
cube.set_sev(met0, cnode2, thrd0, 4);
cube.set_sev(met0, cnode2, thrd1, 4);
cube.set_sev(met1, cnode0, thrd0, 1);
cube.set_sev(met1, cnode0, thrd1, 1);
cube.set_sev(met1, cnode1, thrd0, 1);
cube.set_sev(met1, cnode1, thrd1, 1);
cube.set_sev(met1, cnode2, thrd0, 1);
cube.set_sev(met1, cnode2, thrd1, 1);
cube.set_sev(met2, cnode0, thrd0, 1);
```

```
  cube.set_sev(met2, cnode0, thrd1, 1);
  cube.set_sev(met2, cnode1, thrd0, 1);
  cube.set_sev(met2, cnode1, thrd1, 1);
  cube.set_sev(met2, cnode2, thrd0, 1);
  cube.set_sev(met2, cnode2, thrd1, 1);

  // Output to a cube file
  ofstream out;
  out.open("example.cube");
  out << cube;
}
```

3 Appendix

3.1 File format of statistics files

Statistic files (for an example see3.1) are simply text files which contain the necessary data. The first line is always ignored but should look similar to that in the example as it simplifies the understanding for the human reader. *All values in a statistic file are simply separated by an arbitrary number of spaces*. For each pattern there is a line

```
PatternName  MetricID Count  Mean   Median  Minimum  Maximum   Sum Variance Quartil25 Quartil75
LateBroadcast       6     4 0.010 0.000031 0.000004 0.042856 0.042 0.000459
- cnode: 5 enter: 0.245877 exit: 0.256608 duration: 0.042856

WaitAtBarrier      18    20 0.018 0.006477 0.000002 0.065293 0.369 0.000698  0.000040  0.047409
- cnode: 14 enter: 0.192332 exit: 0.192378 duration: 0.000100
- cnode: 12 enter: 0.326120 exit: 0.335651 duration: 0.065293

BarrierCompletion  17    20 0.000 0.000005 0.000002 0.000018 0.000 0.000000  0.000003  0.000009
- cnode: 14 enter: 0.192332 exit: 0.192378 duration: 0.000009
- cnode: 12 enter: 0.159321 exit: 0.165005 duration: 0.000018

WaitAtIBarrier     27   144 0.001 0.000027 0.000001 0.028451 0.212 0.000028  0.000002  0.000437
- cnode: 11 enter: 0.297292 exit: 0.297316 duration: 0.000057
- cnode: 10 enter: 0.322577 exit: 0.332093 duration: 0.028451
```

Figure 3.1: An example of a statistic file

which contains at least the pattern name (as plain text *without spaces*), its corresponding metric id in the CUBE file (integer as text) and the count -- i.e., how many instances of the pattern exist (also as integer). If more values are provided, there have to be the mean value, median, minimum and maximum as well as the sum (all as floating point numbers in arbitrary format). If one of these values is provided, all have to. The next optional value is the variance (also as a floating point number). The last two optional values of which both or none have to be provided are the 25% and the 75% quantile, also as floating point numbers.

If any of these values is omitted, all following values have to be omitted, too. If for example the variance is not provided, the lower and the upper quartile must not be provided

either.

In the subsequent lines (there can be an arbitrary number), the information of the most severe instances is provided. Each of these lines has to begin with a minus sign (-). Then the text *cnode:*, followed by the cnode id of this instance in the CUBE file (integer as text) is provided. The same holds for enter, exit and duration (floats as text).

The begin of the next pattern is indicated by a blank line.

Bibliography

[1] Message Passing Interface Forum: *MPI: A Message Passing Interface Standard*, June,1995, http://www.mpi-forum.org 1

[2] OpenMP Architecture Review Board: *OpenMP Fortran Application Program Interface --- Version 2.5*, May,2000 http://www.openmp.org 1

[3] K. L. Karavanic and B.Miller, *A Framework for Multi-Execution Performance Tuning*, Parallel and Distributed Computing Practices, 4(3), 2001, September 2

[4] F.Song and F.Wolf and N.Bhatia and J.Dongarra and S.Moore, *An Algebra for Cross-Experiment Performance Analysis*, Proc. of ICPP 2004, 63-72, 2004, August, Montreal, Canada 2

[5] F.Wolf and B.Mohr and J.Dongarra and S.Moore, *Efficient Pattern Search in Large Traces through Successive Refinement*, Proc. of the European Conference on Parallel Computing (Euro-Par), August - September, 2004 Lecture Notes in Computer Science, Springer,Pisa, Italy, 39

[6] J.Labarta and S.Girona and V.Pillet and T.Cortes and L.Gregoris, *DiP: A Parallel Program Development Environment*, Proc. of the 2nd International Euro-Par Conference, Springer, 665-674 Lyon, France, August, 1996 41

[7] Barcelona Supercomputing Center, *Paraver: Obtain Detailed Information from Raw Performance Traces*,Oct,2008, http://www.bsc.es/plantillaA.php?cat_id=485 41

[8] H.Brunst and W.E.Nagel, *Scalable Performance Analysis of Parallel Systems: Concepts and Experiences* Proc. of the Parallel Computing Conference (ParCo), 2003, Dresden, Germany 41

[9] Technical University Dresden, *Vampir - Performance Optimization*, Oct, 2008 http://vampir.eu/ 41

[10] World Wide Web Consortium, *Extensible Markup Language (XML) 1.0 (Second Edition)*, October, 2000 http://www.w3.org/TR/REC-xml 63

[11] Sameer S. Shende and Allen D. Malony, *The TAU Parallel Performance System*, International Journal of High Performance Computing Applications,20(2), 287--331 SAGE Publications, Summer, 2006 52

[12] The Scalasca Development Team scalasca@fz-juelich.de, *Cube 4.2.0- Cube Derived Metrics*, Usage and syntax documentation 22, 62, 64